NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

NPSNETIV: AN OBJECT-ORIENTED INTERFACE FOR A THREE-DIMENSIONAL VIRTUAL WORLD

by

Christopher B. McMahan

December 1994

Thesis Advisors: David R. Pratt
John S. Falby

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1994	Master's Thesis

4. TITLE AND SUBTITLE
NPSNETIV: AN OBJECT-ORIENTED INTERFACE FOR A THREE-DIMENSIONAL VIRTUAL WORLD

5. FUNDING NUMBERS

6. AUTHOR(S)
McMahan, Christopher Brian

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

All simulations require some type of user interface to be functional. The problem is that this interface is sometimes written as a tightly coupled part of the simulation itself, buried in the main body of code. This makes the interface very difficult to create, and once created, more difficult to alter. Thus these tightly coupled simulations do not support a simple way to prototype new interface models and incorporate them into the application.

The solution is to segregate the interface from the main simulation in a loosely coupled system. Communication with the application is through a well-defined interface. By separating the two, changes can easily be incorporated into either module without significant impact on the other.

The approach taken was to create the interface as a separate program. Communication with the main simulation is through data structures passed over a socket communications link. The resulting application displays a control panel on a separate monitor. Using object-oriented techniques, components are easily defined, incorporated, rearranged, and deleted from the panel as needed. The loose coupling to NPSNET means that changes made to one application have little effect on the other. New interfaces can now be quickly and easily created without effecting the core NPSNET application to any significant degree.

14. SUBJECT TERMS	15. NUMBER OF PAGES
NPSNET, Object-oriented, Interface, Motif, ViewKit, Virtual Reality (VR), Simulation, ModSAF	72
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

NPSNETIV: AN OBJECT-ORIENTED INTERFACE FOR A THREE-DIMENSIONAL VIRTUAL WORLD

Christopher B. McMahan
Lieutenant, United States Navy
B.M., Miami University, 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1994

Author: Christopher B. McMahan

Approved By: David R. Pratt, Thesis Advisor

John S. Falby, Thesis Advisor

Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

All simulations require some type of user interface to be functional. The problem is that this interface is sometimes written as a tightly coupled part of the simulation itself, buried in the main body of code. This makes the interface very difficult to create, and once created, more difficult to alter. Thus these tightly-coupled simulations do not support a simple way to prototype new interface models and incorporate them into the application.

The solution is to segregate the interface from the main simulation in a loosely coupled system. Communication with the application is through a well-defined interface. By separating the two, changes can easily be incorporated into either module without significant impact on the other.

The approach taken was to create the interface as a separate program. Communication with the main simulation is through data structures passed over a socket communications link. The resulting application displays a control panel on a separate monitor. Using object-oriented techniques, components are easily defined, incorporated, rearranged, and deleted from the panel as needed. The loose coupling to NPSNET means that changes made to one application have little effect on the other. New interfaces can now be quickly and easily created without effecting the core NPSNET application to any significant degree.

TABLE OF CONTENTS

I. INTRODUCTION

A. BACKGROUND

The role of computer simulations in the military, and specifically in military training, is taking on an ever increasing significance. The cost savings, safety, and convenience are all factors promoting the use of computer simulations to augment traditional tactical and battlefield training exercises. As these simulations gain acceptance and functionality, and the scope of their use broadens, the issue of the human computer interface also takes on much more significance.

Just as important, though, is the issue of software design. With the increasing power of computer systems, and the increasing capabilities of the simulation packages available, a program can soon take on monstrous proportions. Steps must be taken to ensure that the code can be easily maintained, modified, and expanded as necessary. This research is an effort in that direction. Specifically, this paper will describe research in the area of interface design for a virtual world, NPSNET, developed at the Naval Postgraduate School [ZYDA93].

B. MOTIVATION

1. The Need for a User Interface

NPSNET has accumulated a great deal of functionality during the course of its many iterations never dreamed of at its inception. As the scope of the research expands, new applications for the NPSNET technology continue to emerge. Tank and tracked vehicles, missiles, rockets, small arms, ships, and even individual foot soldiers have been added to the environment.

These enhancements come at a price however. Since the focus of NPSNET research has been to explore the "proof of concept" of the networked virtual world, the user interface

has not been the driving factor. Up to a point, the simple key strokes and input device combinations have been sufficient to operate within the environment. With each upgrade to NPSNET, however, the interface between the virtual world and the participants becomes a more pressing issue.

Now that the proof-of-concept phase has in large part passed, and the real-world applications of the NPSNET technology are just beginning to be exploited, the means to create logical, consistent interfaces must be developed. Within the confines of the NPS Graphics and Video Laboratory, the users have been largely limited to the relatively homogeneous group of students, faculty, and interested military and academic specialists. The move to real world applications, however, brings along with it a much wider range of user sophistication.

The pressing need for an interface is evident when the environment is demonstrated to local officials and school children. Time that could be spent teaching and discussing the relative merits of NPSNET, virtual worlds, networked environments, simulations, and computer technology in general is instead spent continuously reviewing key commands and explaining how a head's up display works.

2. Expansion of NPSNET capabilities

NPSNET is an experiment in networking a large number of users interacting within a single virtual environment. By nature, this implies that the system must be capable of supporting a wide, disparate group of participants. Some will be foot soldiers, others tanks, aircraft, artillery, and exercise observers. To accommodate these many interface requirements, several lines of thinking have been pursued.

The first implies building the necessary interface components to support all possible scenarios within the framework of NPSNET itself, along with the simulation engine to generate vehicle dynamics and movement, and the visualization system to display the world and its participants. This monolithic approach (Figure 1) requires the builders of the environment to anticipate every possible application of the program, and build the

appropriate interface, or face frequent additions to the basic code, along with the additional overhead of reduced efficiency, multiple compiles, and immediate obsolescence as requirements change. This route also deters the researchers from the primary goal of building the environment and its network capabilities by focusing their efforts on the many interface requirements. This is the approach that has been taken with all preceding versions of NPSNET.

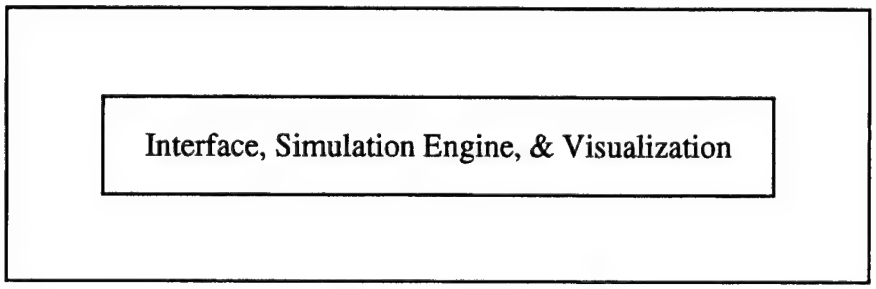

Figure 1: Monolithic Model of Virtual World Simulators

The second paradigm is to completely separate the virtual environment from the interface. This is accomplished in the case of NPSNET by building interface "modules" or panels that operate outside of NPSNET code, and communicate with the environment itself. The interface in this new NPSNET (Figure 2) provides a layer of insulation between

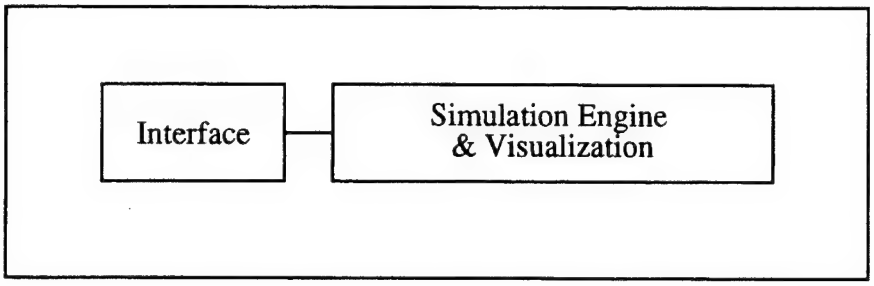

Figure 2: New NPSNET Model of Virtual World Simulators

the user and the virtual environment. Changes to the environment can then be easily incorporated into the interface modules just by rewriting the interface components.

Additionally, as new interface ideas are explored, the required modules could be incorporated without any danger of corrupting the efficiency or code of NPSNET itself.

The almost complete separation of NPSNET from the interface components has the added benefit of allowing future designers to incorporate physically-based flight dynamics, vehicle movement, artillery fire, fuel, weapons performance and such into the interface code rather than the NPSNET code (Figure 3). This greatly relieves the burden on the

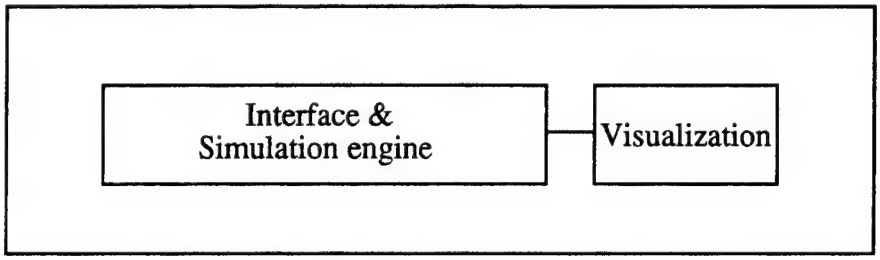

Figure 3: Proposed Future Expansions of NPSNET Model

environment by distributing the workload to the interface modules. If a tank runs out of fuel, the interface itself notifies the driver, and renders the tank stationary. The only change registered within the network is the tank has stopped moving. No overhead is required to notify the entire net that the tank is out of fuel. No global variables for fuel state of each player are required. This allows the designers of each participating system to create and tailor vehicles with the appropriate interface reflecting the capabilities and limitations of that vehicle without major modification of the NPSNET code itself.

The next logical step is the additional separation of the simulation engine from NPSNET as well (Figure 4) This model has the added advantage of allowing designers to customize vehicle parameters as easily as the interface, allowing for modifications in performance characteristics, and even experimentation with new vehicle and weapon designs within the virtual world. The disadvantage of this system is the extra computing power required to run each engine separately. The ModSAF program [LORA94] currently employs this model, requiring the visualization, simulation engine, and interface to run on separate computers.

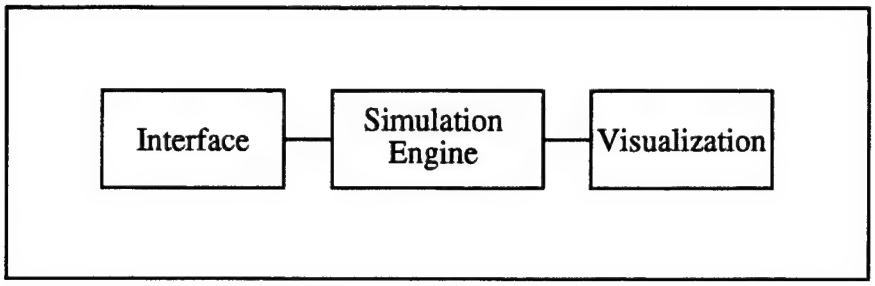

Figure 4: Complete Separation of Virtual World Simulator Engines

C. OBJECTIVES

The objective of this research is the design and construction of an application framework to enable the rapid creation and incorporation of interface modules for NPSNET. This framework provides the capability for reusable interface components that can be "plugged" into modules, and yet offers the flexibility to customize these components, and build new ones as required. Another objective is the specification of a standard method of communication between the interface modules and NPSNET.

D. ORGANIZATION

Chapter II provides an overview of the NPSNET project, including its purpose, history, and general program design. Chapter III describes the interface design. Chapter IV describes each component of the interface control panel. Chapter V discusses the class hierarchies used. Chapter VI details the method of communication employed by the interface program. Chapter VII presents the results of the interface framework, and a description of using the system. Chapter VIII presents the conclusions of this research and recommendations for follow on work.

II. OVERVIEW OF NPSNET

Since 1990 students and faculty at the Naval Postgraduate School have been working on an ongoing research effort into the feasibility and applications of a low-cost, real-time virtual world simulation system. This system, known as the Naval Postgraduate School Networked Vehicle Simulator (NPSNET) operates on the Silicon Graphics IRIS family of graphic workstations, utilizing the Distributed Interactive Systems (DIS) network protocol as the interface for sharing information across the network [ZYDA93].

NPSNET is primarily a research testbed used to provide the students of the Naval Postgraduate School experience in the potential benefits and pitfalls of networked virtual environments. Students participating in this research bring their own special interests and areas of concentration, making diverse contributions to the project. Participants gain the insight, knowledge, and experience of working with a large project, learning the specific capabilities and limitations of interactive virtual environments. As an ongoing research project, therefore, NPSNET continues to grow and evolve. Recent additions include the use of spacialized three-dimensional sound, weather modelling, smoke effects, and the use of articulated individual humans within the virtual environment.

The NPSNET research focuses primarily on the application of virtual worlds. Where other efforts in virtual environments have emphasized the hardware input and interaction within a simulated environment, NPSNET has concentrated on the software side of the equation. Emphasis has been placed on areas such as the incorporation of the latest DIS protocol, the cost-efficient use of off-the-shelf components whenever possible, incorporation of physically-based modelling, dynamic terrain features, and autonomous forces and expert systems [ZYDA93].

NPSNET is written using object-oriented design principles in C++. Entity characteristics are based on class inheritance, where the subclasses take on more

specialized characteristics of the parent class. For example, a class Vehicle has the inherited classes Air, Stealth, Ship, Submersible, and Ground. These classes, in turn, may have additional inherited classes as well (Figure 5). This object-oriented design allows for the relatively easy definition of new types of vehicles and weapons. A new system can be designed by specifying additions to an already defined class of entities. Thus a new fixed-wing aircraft can inherit all of the characteristics of the current fixed wing class, and only specify additions and differences that define the new characteristics of the system. There is no need to redefine all of the characteristics of a fixed wing aircraft, since these are already given by the parent class.

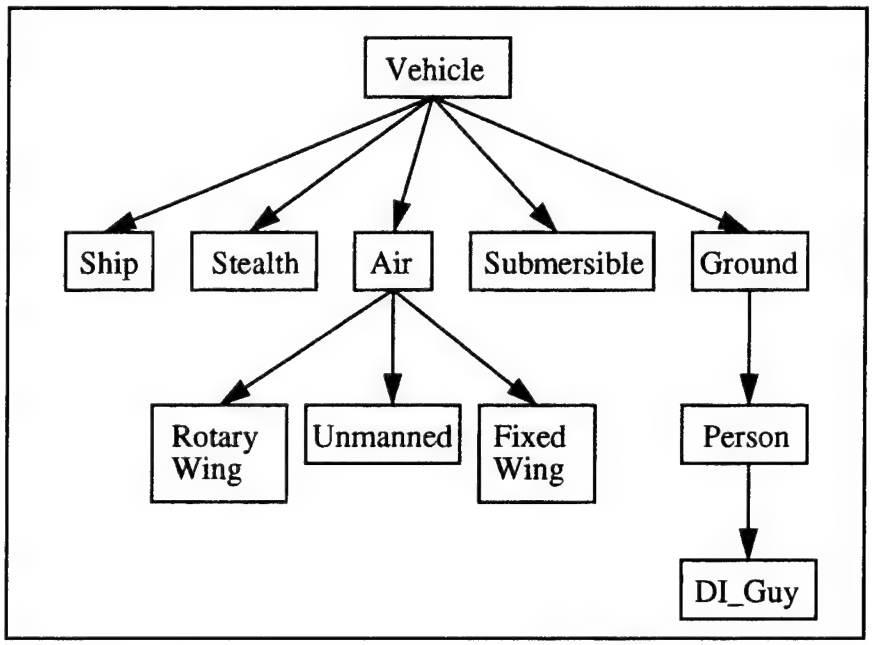

Figure 5: Class Hierarchy in NPSNET

In the past, NPSNET has provided three options for computer-human interface: SpaceBall, flight control sticks, and the keyboard. These input options can be selected during execution of the simulation from the keyboard, allowing the user to switch dynamically as input requirements change.

The Spatial Systems SpaceBall is a spherical knob which affords the user six degrees of freedom. Unfortunately, it is very difficult to use this device effectively, since no single degree of freedom can be easily isolated. When the user tries to accelerate, for example, a very slight forward movement may also result in a forward rotation on the ball. This results in a nose-down maneuver of the vehicle.

As a result of the difficulty associated with the SpaceBall, flight control sticks were incorporated into the simulation. These consist of the familiar joystick and throttle stick known to all pilots and video game players. Thus a familiar paradigm could be used to allow players to easily concentrate on the simulation itself. The flight control sticks also allow a much easier isolation of the degrees of freedom, making movement in the world a much more intuitive process [ZYDA93].

The difficulty with both the SpaceBall and flight control sticks, however, is the limited amount of input afforded to the user. To change any environmental parameters within NPSNET (time of day, scene drawing options, tethering, etc...) a keyboard remains another selectable option for input. Where the keyboard excels in providing access to all options of NPSNET, it suffers in vehicle movement, and familiarity with the command set. With so many options available to the user, learning all of the possible key sequences can be a daunting task at best.

III. ARCHITECTURE OF THE INTERFACE

A. MOTIF AND C++

Motif is based on the Xt Intrinsics, a library that supports an object-oriented architecture implemented in C. The objects supported by Xt and Motif, however, are completely different than the C++ objects. Motif uses the object structures internally, but still presents a programming interface that is functional in nature. Using Motif is no different than using any other C language library.

Since C++ was designed to allow programmers to continue to use the C libraries, programmers can continue to take advantage of the existing Motif libraries while employing object-oriented programming techniques. This permits the use of object-oriented programming techniques to build interface components that can be further developed and combined to create increasingly sophisticated interfaces.

The primary reason to develop object-oriented Motif components is to enable the programmer to build a library of pre-existing, pre-tested interface components that can be combined to form complete applications. This library encompasses an applications framework that defines not only the components available, but also the structure and control flow within each component, an interface to other components, and ultimately, the structure of an entire application.

B. VIEWKIT DEVELOPMENT TOOLKIT

The interface panel was written using an application framework developed by Doug Young [YOUN93] which allows encapsulation of Motif widgets into C++ classes. The Viewkit system, a part of the Silicon Graphics Irix 5.2 operating system, is a C++ toolkit that provides a collection of high-level user interface components, such as windows, menus, and dialog boxes [VIEW94]. All components in ViewKit are C++ classes, which provide the basic framework for using the Motif widgets in a structured, object-oriented

manner. The portion of the ViewKit system used in the interface panel is a small subset of the entire capabilities of the toolkit. The most important components used are the VkComponent class, the VkApp class, VkWindow class, and the VkMenu class.

The VkComponent class provides an abstract base class to define a structure and encapsulate a collection of widgets. It provides an access function to a component's top level widget, handles the destruction of encapsulated widgets, and provides access functions for accessing some of the class's data members.

The VkApp class, derived from VkComponent, handles the initialization and event-handling operations common to all Xt-based applications, such as XtAppInitialize and XtAppMainLoop. The VkApp class also provides support for functions such as quitting the application and showing, hiding, iconifying, and opening the application's windows. Additionally, the VkApp class stores essential information such as XtAppContext, a pointer to the X display structure, and the application and class names.

The window handling routines consists of two classes, VkSimpleWindow and VkWindow, derived from the VkComponent class. These classes are used to create and manipulate the top-level windows in a ViewKit application. VkSimpleWindow supports a top-level window that does not include a menu bar. The VkWindow, derived from the VkSimpleWindow, adds support for a menu bar along the top of the window.

The VkMenu class provides easy creation of menu bars, menu panes, popup menus, option menus and cascading menu panes. This package also provides facilities for activating and deactivating menu items and dynamically adding, removing, or replacing menu items and menu panes.

C. INTERFACE APPLICATION STRUCTURE

The interface panel is built from lower level components and widgets which combine to form larger components. These components are then instantiated within a main window component called Panel. Wherever possible, class names correspond to actual panel components (Figure 6).

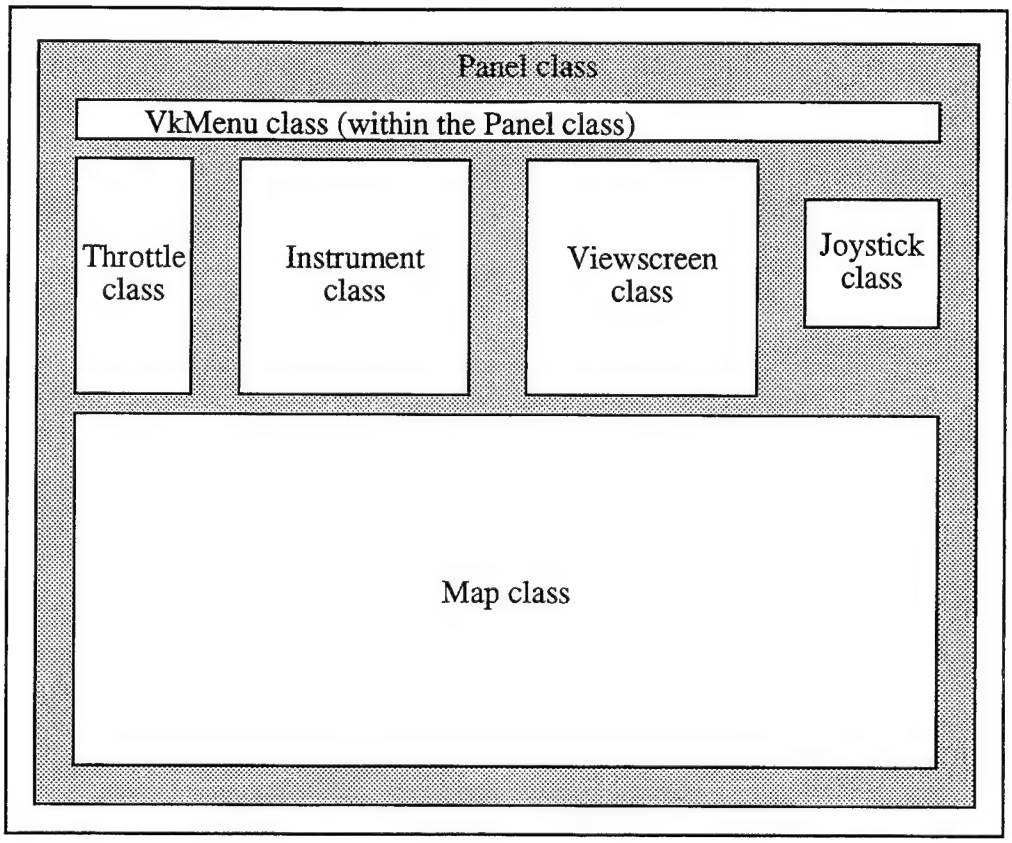

Figure 6: Main Component Classes within the Panel

Communications between the panel and the host computer running NPSNET are done through a defined data structure passed over a broadcast mode socket communications stream. This data structure is designed to be flexible enough to meet any future demand for interface design and program requirements. Information is also brought in directly from the NPSNET simulation DIS packets. Outside support library functions are provided by the ModSAF program [LORA94] and the NPSNET network library [ZESW93].

IV. CLASS HEIRARCHIES

The class hierarchy within the interface panel is very simple. All components of the interface are derived from the VkComponent with the exception of the Joystick class, the Panel class, and the Message class (Figure 7).

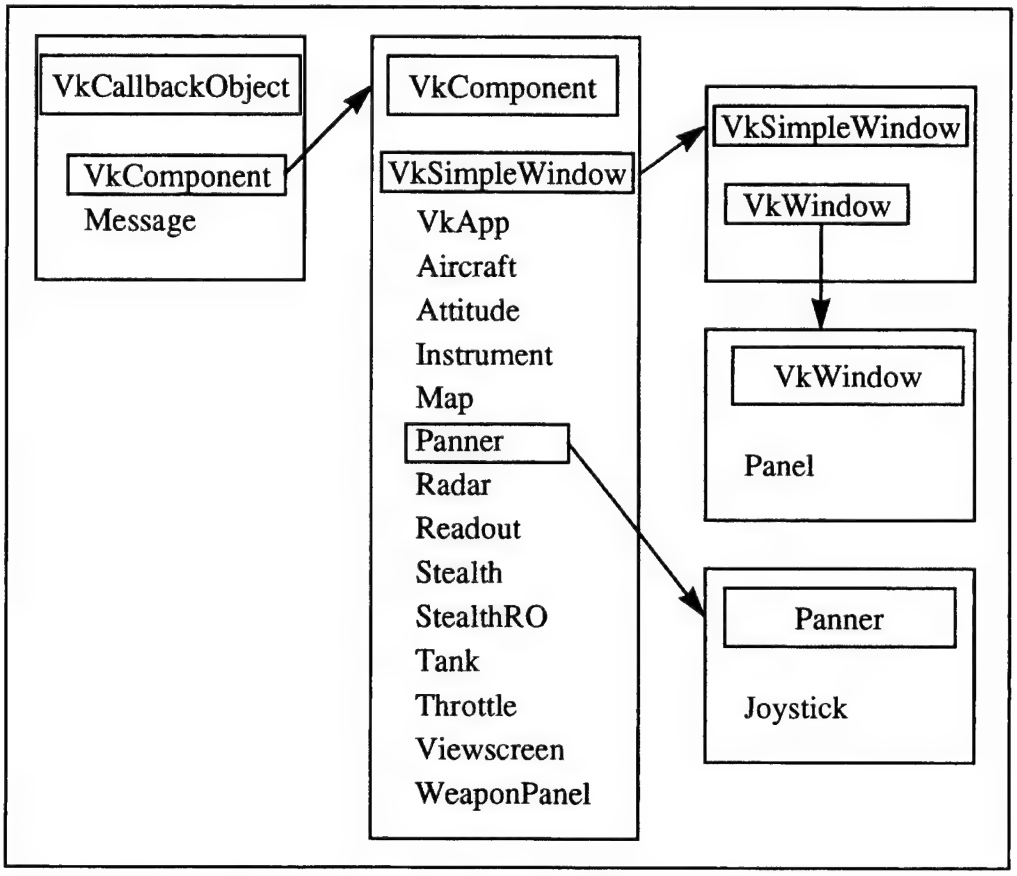

Figure 7: Class Hierarchies within the Panel

The VkCallBack is the top level class within the ViewKit toolkit. It provides the basic structures necessary to encapsulate the components within an interface, but does not directly support the creation of widgets as derived classes. The message class is derived from VkCallBack rather than VkComponent since it does not require any support for Motif

widgets. The primary functions of the Message class are to serve as the central distribution point for incoming and outgoing traffic, and provide support for the timer (VkPeriodic class) which executes the readSocket code.

The VkComponent, derived from VkCallBack, is the top level class that supports interface components. The VkComponent class provides a function to access the top level widget within any derived component. This function, baseWidget(), is necessary to set the constraint resources of the widget when placing it within a larger component. It returns a type widget, and can be used any place that date type is required. For example, the following code segment, taken from the file panel.C, illustrates the use of the baseWidget function to position the throttle component within the interface panel itself (Figure 8).

```
// position the throttle component within the panel
XtVaSetValues(throttle->baseWidget(),
   XmNtopAttachment, XmATTACH_FORM,
   XmNtopOffset, margin,
   XmNleftAttachment, XmATTACH_FORM,
   XmNleftOffset, margin,
   NULL);
```

Figure 8: Use of the VkComponent baseWidget Function

The joystick is derived from the Panner class. The Panner class sets up the joystick widget with the drawn area, the drawn button, and functions for button movement, mouse events, and output. The joystick class overrides the drag function to translate the coordinates into a format suitable for NPSNET. New functions added were the release function, which snaps the joystick to the center position upon release of the mouse button, and the positionToJoystick, which converts the raw joystick screen coordinates into the format required by NPSNET, with a sensitivity factor built in to de-sensitize the center area of the frame.

The Panel is derived from the VkWindow class because it is the top level window within the interface application. The VkWindow class is one of two class structures

designed to accommodate top level windows (Figure 9). It is actually derived from the

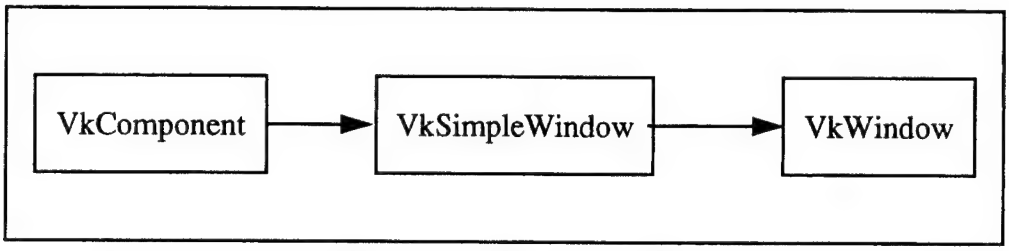

Figure 9: Class Hierarchy of Top Level Window Managers

other top level window class, VkSimpleWindow. VkSimpleWindow class supports the necessary functions for window creation and event handling, but with no provisions for a menu bar. VkWindow adds functions and support for menus.

V. COMPONENT DESCRIPTIONS AND FUNCTIONS

A. READOUT COMPONENT

The readout is the least complex component within the interface panel. It consists of a textfield widget and a label (Figure 10). The justification for creating a component at a level so close to pure Motif code is that every occurrence of a digital display of any kind within the interface is invariably accompanied by a label describing what the information represents. Thus, rather than repeat the cycle of textfield/label for every readout, a readout class was created to handle this mundane task automatically. This is the essence of this development system, encapsulating components within class structures for easy reuse within an interface.

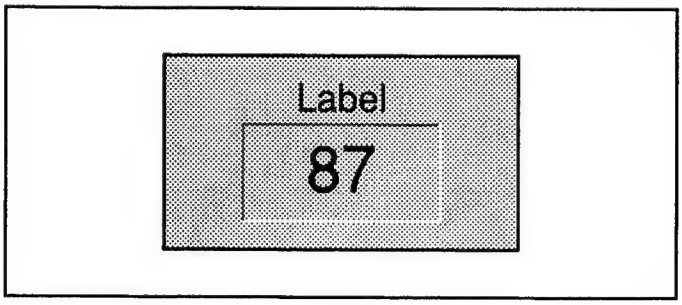

Figure 10: The Readout Component

The readout class has the following access functions:

setLabel(char*) Takes a pointer to a string and uses that string to set the label of the readout instantiation.

setLeftOffset(int) Determines how far in pixels the readout is to be offset to the left.

setTopOffset(int) Determines how far in pixels the readout is to be offset from the top.

setDisplayValue(int) Used to set the value of the information displayed in the readout based on incoming socket traffic from NPSNET. Each class that instantiates a readout will

use this command to set the values of their specific instantiations of a readout class, so the message class never addresses the readout functions directly. This function is also used upon instantiation of the readout classes in components to set the initial display value.

setDisplayValue(float) Overrides the setDisplayValue to enable the readout to display float values automatically if float is used as an input rather than integers.

Instantiating the readout class (and any other component class within the interface) within a larger component merely involves creating a new instance of a reader class within the definition of the parent class. The component is then attached and placed within the larger component by accessing the VkComponent function baseWidget(), which returns the component's top level widget (Figure 11).

```
// the throttle thrust readout goes here
// use an instantiation of the readout class
thrust = new Readout("thrust", _baseWidget);

XtVaSetValues(thrust->baseWidget(),
    XmNtopAttachment, XmATTACH_FORM,
    XmNleftAttachment, XmATTACH_FORM,
    NULL);

// assign a text string to the label readout class using the
// class member function setLabel
thrust -> setLabel("% Thrust");

// set the initial value of the thrust window
thrust -> setDisplayValue(0);

// instantiate the thrust component
thrust -> show();
```

Figure 11: Instantiation of a Readout Class Component

B. ATTITUDE INDICATOR

The attitude class is an encapsulation of the OpenGL program which draws the attitude indicator. OpenGL, which stands for Open Graphics Library, is a software interface to the graphics hardware. This library provides the programmer with approximately 120 commands that specify objects and operations needed to produce interactive three-dimensional graphics [NEID93]. The Attitude indicator, familiar to pilots, is an instrument

which indicates the pitch and roll of an aircraft by displaying an artificial horizon, with tick marks to measure the pitch and roll angles (Figure 12).

The attitude class is derived from VkComponent, and provides a single command to the user, which in this case is the aircraft class. The attitude class places the results of the GL calls into a GlxMDraw widget, which provides a structure into which the graphics routines can be rendered.

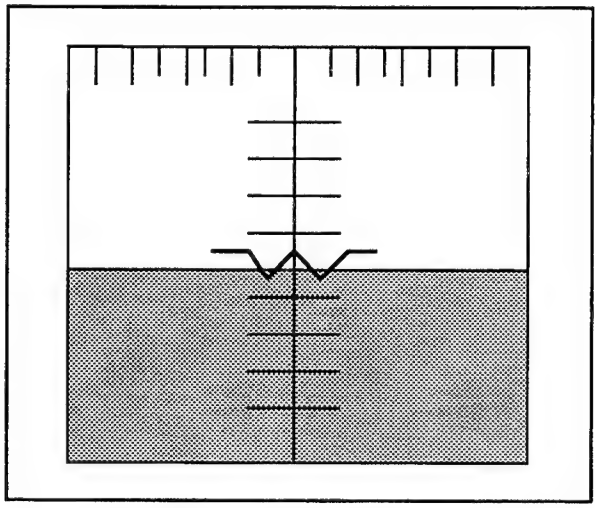

Figure 12: The Attitude Indicator

The attitude class provides the user the following command:

draw_horizon(float pitch, float roll) draws the artificial horizon with the appropriate roll and pitch using GL programming calls. The attitude indicator panel (horizon, roll and pitch indices, ground, and sky) are rotated along the Z-axis by the given roll angle, then translated along the Y-axis by the opposite of the pitch angle. The opposite pitch is used since the pitch angle provided represents the pitch of the aircraft, and the attitude indicator displays the relative position of the ground. As the pitch of the aircraft increases, the nose of the aircraft point up, and the relative angle of the ground decreases.

C. WEAPON COMPONENT

The weapon class provides a simple means of assigning interface buttons to any platform's weapons system. It is a flexible way of labeling the appropriate buttons, and

setting up a unified layout of the weapons assembly. Upon instantiation, an additional argument is passed in which indicates the number of weapons, up to three, assigned to a given platform. The appropriate layout is then set up, and callbacks to activate each button are initialized (Figure 13).

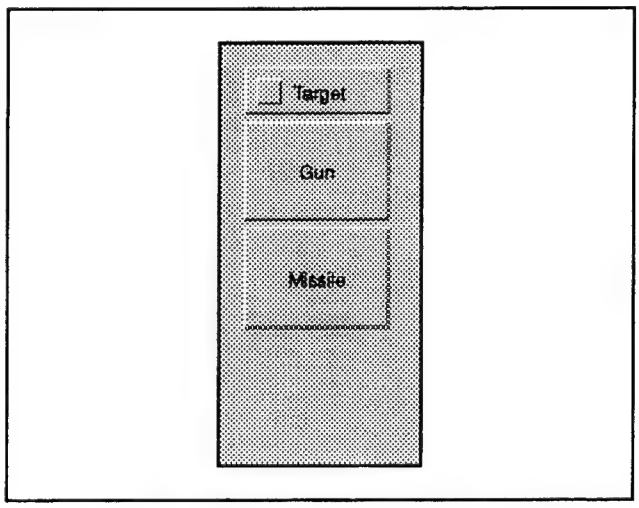

Figure 13: The Weapons Component with Two Weapons

The functions provided with the weapons class are as follows:

setPrimaryLabel(char *)

setSecondaryLabel(char *)

setTertiaryLabel(char *) sets the labels that appear on the primary, secondary, and tertiary buttons firing buttons.

targettingButtonState(short) passes the current state of the targeting toggle button (on or off) to the message class.

primaryWeaponState(short)

secondaryWeaponState(short)

tertiaryWeaponState(short) send the state (on or off) of the appropriate firing button. When a mouse button is pressed over a firing button, the callback for that button is executed, which executes the corresponding WeaponState function with a value of TRUE. If a mouse button is released anywhere after being pressed within the firing button, the

corresponding WeaponState reverts back to FALSE, causing the weapon affected to stop firing.

D. RADAR CLASS

The radar class encapsulates the GL program which draws the radar screen. It is used as a component to build the interface viewscreen. The reason that two levels of class structure are used in the viewscreen component is to enable the future incorporation of functions such as camera views, infra-red and thermal images, and light-intensifying images into the viewscreen component.

The radar is rendered into a GlxMDraw widget, just as the attitude indicator. Once the graphics for the GlxMDraw widget are set up, the timer within the message class is started by executing the command

```
theMessage -> _timer -> start();
```

The timer is started within the radar class to ensure that the graphics context for the GL window is completely set up first. The timer looping function includes a call to update the radar screen with the incoming DIS traffic. If this call were to be executed prior to the complete initialization of the GL window, a segmentation fault would result, since rendering was attempted to a non-existent widget.

Once the GL window is set up, and the update function is executing, the radar images are drawn onto the radar screen as letters from a custom-designed icon font. Each letter in this font, a standard font used with the NPSNET code, represents a different kind of vehicle, from a fixed-wing aircraft to a tank to an individual footsoldier. The radar class provides the following functions to the user:

updateRadar(float posX, float posY, float hdg)

The function updateRadar is one of the functions executed from the message class as a result of the VkPeriodic timer events. Within this function, the window into which the GL code is to be rendered is specified with the GLXwinset command. The background color is set. Range rings are displayed at ranges of 1000, 5000, 10,000, and 20,000 meters. The

position of the NPSNET host vehicle is then used to set the center point of the display window using the GL translate function. The host vehicle is always centered in the display, with the front of the vehicle facing the top of the radar screen. The heading parameter is then used to rotate the entire radar screen with the GL rotate command. This places the icons representing the other entities in the correct positions relative to the heading of the host vehicle (Figure 14).

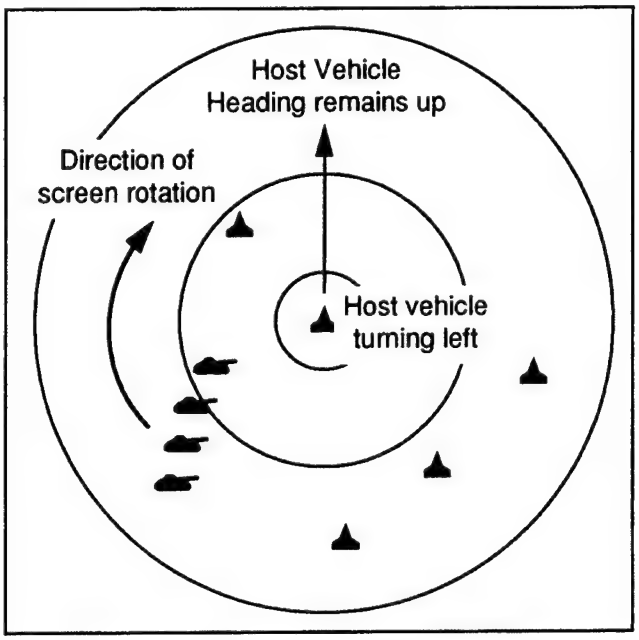

Figure 14: Rotation of the Radar Screen as Host Turns Left.

Within the updateRadar command, a while loop is executed which cycles through an array structure set up in the network library. This array contains pointers to all of the vehicles participating in the exercise. As data on each entity is read, the radar code parses the entity type and location. The location is translated into the window coordinate system, and the appropriate entity icon is displayed in the correct position within the window.

setRange(int range) is called by the viewscreen class as a result of input events on the range scale. The function setRange uses the value passed in as range to calculate the area to display in the radar screen, based on a percentage of total world size (Figure 15). Total world size is used as a basis for calculations, rather than set radar ranges, to enable the radar

screen to display the entire area no matter what the world size. To do this, the setRange function determines the larger axis (X or Y) of the world size. This axis is then used as a basis for computing the amount of range to display on the screen by multiplying with the range input and dividing by 100.

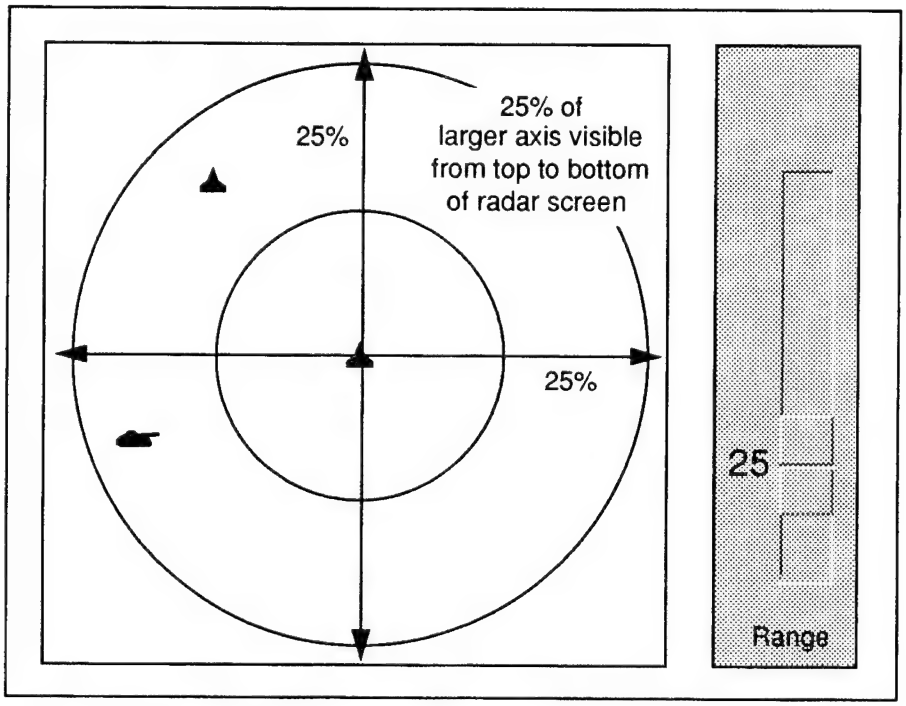

Figure 15: Radar Display with Range Set to 25

An important function, not accessible to the user, is the pickVehicle function. It is invoked as a result of the GlxNinputCallback event. The callback specified by this function normally responds to any user input to the GlxMDrawingArea from the mouse or keyboard. In order to limit response to only button presses of the mouse, the translation table, which defines the input parameters, was rewritten (Figure 16).

The callback function inputCallback then executes the function pickVehicleChanged, in the message class. The function pickVehicleChanged, executed in the timer loop, uses the returned value from the radar function pickVehicle as its input parameter. The function pickVehicle draws an overlay plane onto the OpenGL window and intercepts the mouse input. The position of the pointer is compared with the graphics drawn on the radar screen

```
// replace the default translation table for the
// GlxNinputCallback with a new one that only responds
// to mouse button down events, rather than the default
// of all button and keyboard events.

String translations = "<BtnDown>:glxInput()";

// use the redefined translation tabel in place of the default
XtSetArg(args[count], XmNtranslations,
        XtParseTranslationTable (translations)); count++;
```

Figure 16: Overriding the Default Translation Table

and, if any icon falls within the prescribed "pick region" using the system calls pick and endpick, the index number of the vehicle within the vehicle array is returned (Figure 17).

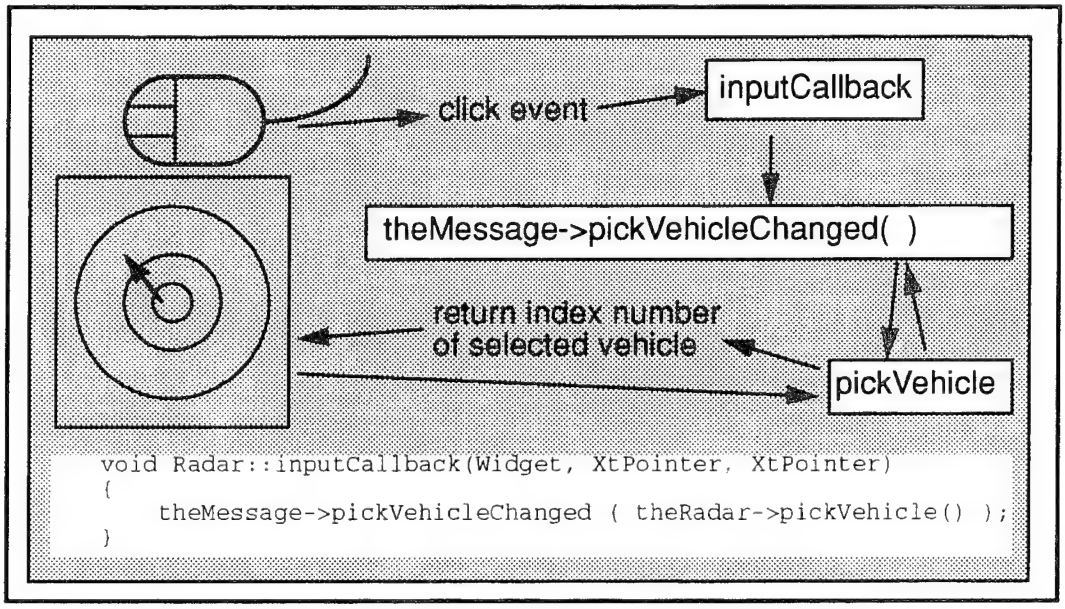

Figure 17: Sequence of Events in Selecting Vehicle

E. VIEWSCREEN COMPONENT

The viewscreen component is instantiated as a class viewscreen within the class panel. It is the parent component for the radar class, consisting of a form widget into which is

inserted a frame assembly, the radar screen, and toggle buttons and a scale widget to control the radar (Figure 18). The viewscreen class supports the following user function:

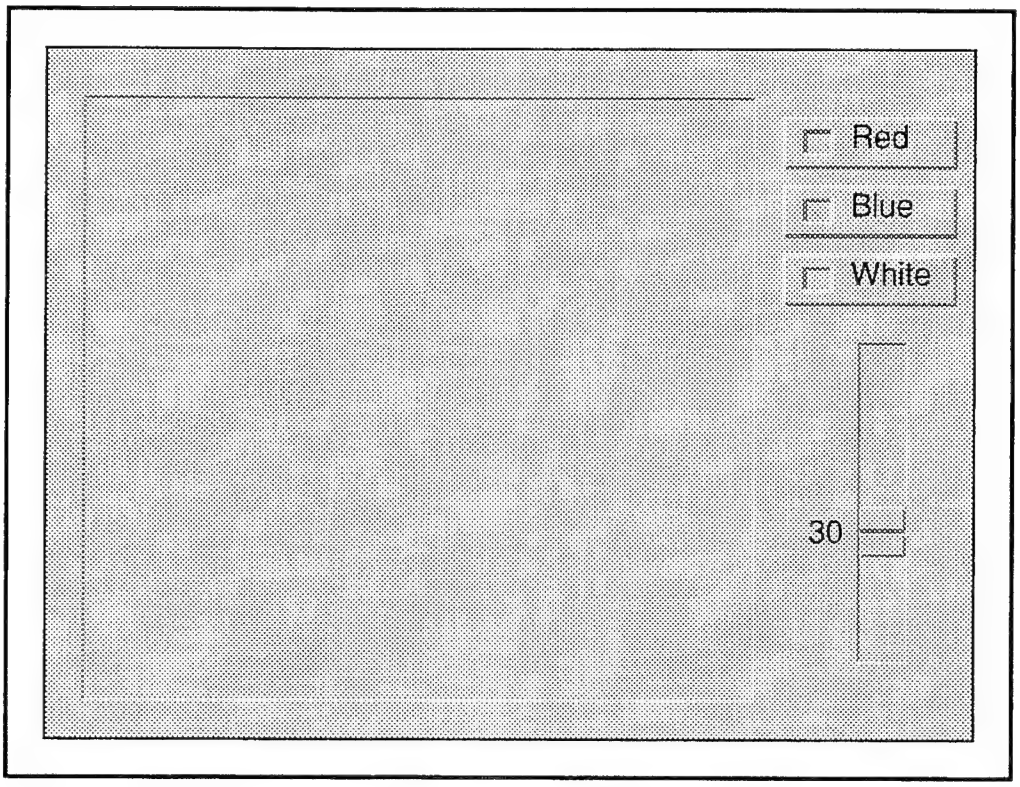

Figure 18: The Viewscreen component

updateScreen(GUI_MSG_DATA) which parses the socket structure parameter and sends the host vehicle's current X coordinate, Y coordinate, and heading to the radar class through the radar function updateRadar.

F. THROTTLE COMPONENT

The throttle is instantiated as a class throttle. It consists of a scale widget and a readout class to display thrust information (Figure 19). The parent class of the throttle is

VkComponent. It is instantiated and managed from within the definition of the panel class, in the file panel.C.

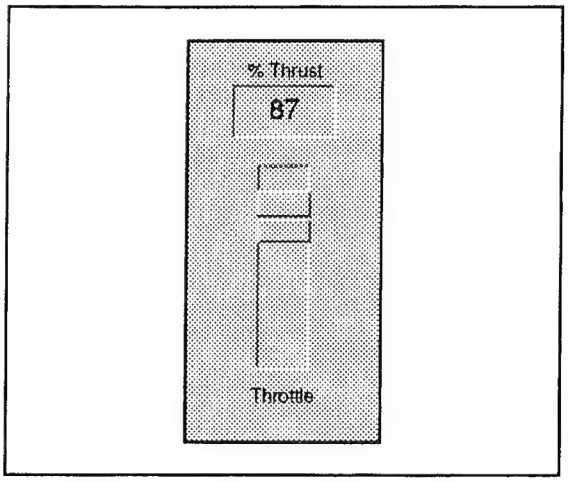

Figure 19: The Throttle Component

The throttle is a user input and an output component. The user input is through sliding the scale to adjust the throttle position, or pressing the left mouse button within the widget itself, which executes the callback functions

The throttle class provides the following functions to the user:

setThrustDisplay(int) sets the value of the thrust display. This function is executed from within the message class based on incoming socket data from NPSNET.

updateThrottle(GUI_MSG_DATA) sets the position of the scale slider based on information received from the socket stream from within the message class. This allows the throttle to reflect the position of any alternate input devices, such as the throttle from the flight control stick assembly.

G. INSTRUMENT COMPONENT

The instrument component is basically a shell that displays information common to all interface components, and supplies a fairly large area in which to instantiate user-selected

vehicle panels (Figure 20). The parent class of the instrument panel is VkComponent, and it is instantiated within the panel class in the file panel.C.

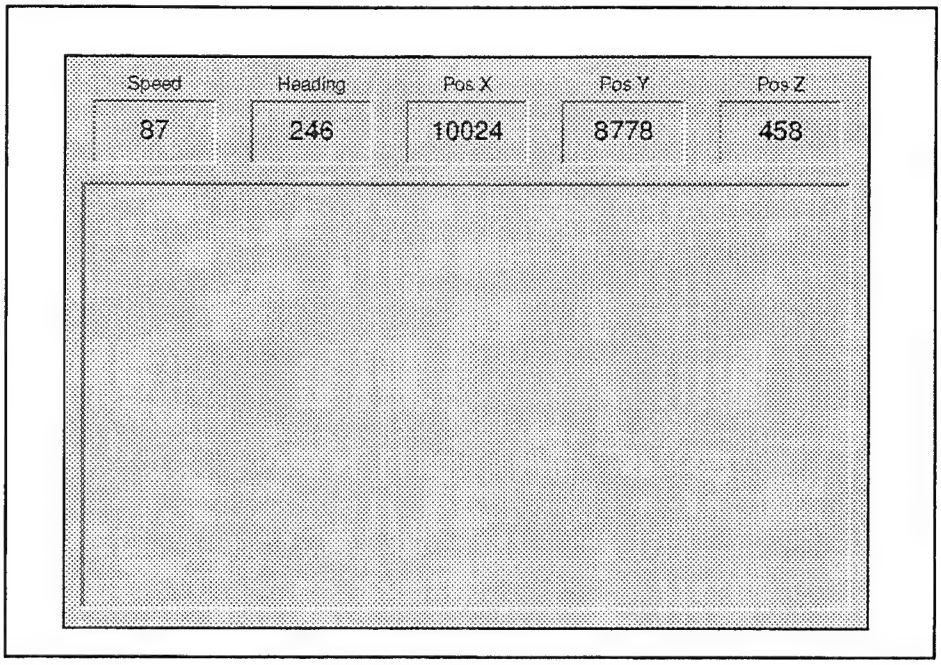

Figure 20: The Instrument Component

The base widget for the instrument is a form widget. A rowcolumn widget contains instantiations of the readout class for speed, heading, X position, Y position, and Z position. The recessed frame below the readout widgets holds a form widget set to dimensions of 450 pixels width by 241 pixels height. It is within this form that the vehicle interface panels are instantiated.

The classes for the vehicle panels are actually created within the instrument class. The only time the show() command is executed to display a vehicle panel is as a result of user input from the panel pull-down menu. The callback for the menu within the panel class executes a function directly in the instrument class which hides the currently displayed panel (if any) and shows the new selected panel. Notable here is the direct communication between the instrument and the panel class, without any routing through the message class. The reasoning behind this departure is the fact that the selection of which panel to display

does not affect any information flowing to or from the interface to NPSNET. If such a need became necessary in the future, it would be a trivial matter to place an intervening function within the message class to pass the menu selection on to the instrument class.

The single user function takes two parameters as input. These are the GUI_MSG_DATA structure and an integer representing the index to a selected vehicle in the radar screen.

updateInstrumentDisplays(GUI_MSG_DATA *, int) This function is executed from the message class within the timer loop. It parses data from the GUI_MSG_DATA structure to update the instrument class readout displays. The current active vehicle is then determined based on the value of the variable _currentPanel, which is global within the instrument class, and executes the appropriate update function (Figure 21).

```
void Instrument::updateInstrumentDisplays(GUI_MSG_DATA *pdu_ptr,
int pickIndex)
{
    // update the readout values along the top of the
    //instrument panel
    speed->setDisplayValue(int(pdu_ptr->velocity));
    heading->setDisplayValue(int(pdu_ptr->heading));
    posX->setDisplayValue(int(pdu_ptr->positionX));
    posY->setDisplayValue(int(pdu_ptr->positionY));
    posZ->setDisplayValue(int(pdu_ptr->positionZ));

    // determine which panel is active and update its display
    switch (_currentPanel)
    {
        case 1: // stealth
            theStealth->updateStealthDisplay(pickIndex);
            break;
        case 2: // aircraft
            theAircraft->updateAircraftDisplay(pdu_ptr);
            break;
        case 3: // tank
            theTank->updateTankDisplay(pdu_ptr);
            break;
        default:
            break;
    }
}
```

Figure 21: Instrument Class Function updateInstrument

As new vehicles are added to the interface, it is a simple matter to add update functions to the instrument class. Determination of the active vehicle is done by setting a global

integer flag (_currentPanel) within the function setNewPanel. This function is executed by the panel class upon user selection of a vehicle from the pull-down menu. Each vehicle is associated with an integer that is passed to setNewPanel from that menu item's callback. The function for hideCurrentPanel first determines which panel is active by checking the current value of the global panel integer. The new panel is then shown using the parameter passed in from the pull-down menu callback, and the global _currentPanel variable is updated (Figure 22).

```
// set the new panel within the instrument cluster.
// Executed from panel class by a pull-down menu selection
void Instrument::setNewPanel(int newPanel)
{
    theInstrument->hideCurrentPanel();
    switch (newPanel)
    {
    case 1: // stealth
        theStealth->show();
        break;
    case 2: // aircraft
        theAircraft->show();
        break;
    case 3: // tank
        theTank->show();
        break;
    }
    _currentPanel = newPanel;
}
```

Figure 22: Setting Active Vehicle Component within the Instrument Class

H. AIRCRAFT PANEL

The aircraft class is instantiated within the instrument class, but is not shown immediately. It is one of three components that can be placed in a single location within the instrument component. It is this location, which also supports the stealth and the tank panels, that sets the specific control functions unique to each vehicle.

The aircraft component consists of three readout components, an attitude component, and a weapons component. Each of these is instantiated within the aircraft class description (Figure 23).

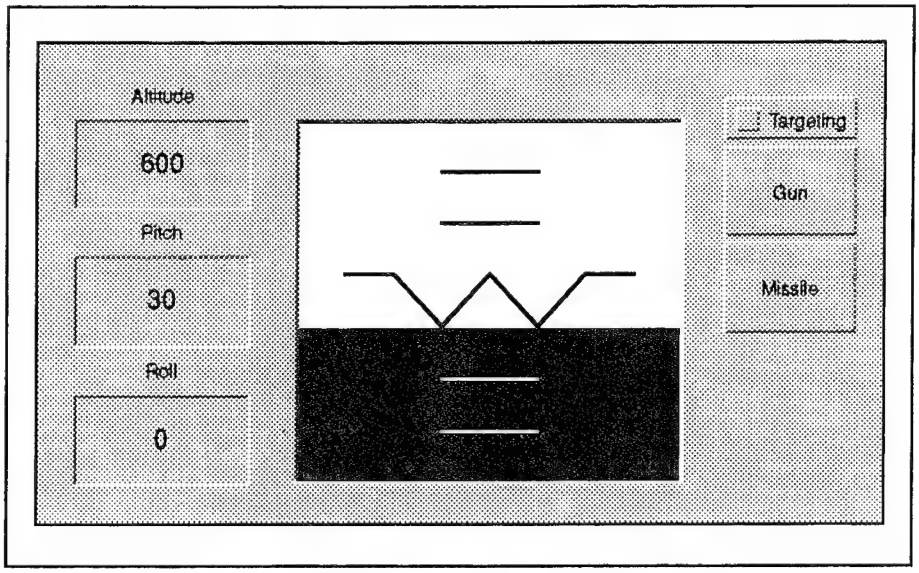

Figure 23: The Aircraft Component

The base widget of the aircraft component is a form widget, set to a width of 450 pixels and a height of 240 pixels. These dimensions allow the aircraft panel to fit into the display area set aside for it in the instrument panel. To this form are attached a row column widget containing the readouts for altitude, pitch, and roll, the attitude indicator, and the weapons panel.

The aircraft class supports the following command:

updateAircraftDisplay(GUI_MSG_DATA *) This command takes in a pointer to the incoming socket structure, extracts the appropriate information, and calls each of the components within it (Figure 24). It is executed within the message class as part of the message class function read_socket().

The updateAircraft command sends the altitude, pitch, and roll heading to the readout class functions setDisplayValue. The next command, GLXwinset, then takes the base widget of the attitude indicator, accessed through the ViewKit function baseWidget() as the

widget into which the GL code is rendered. The actual GL code is then executed to update the attitude indicator with the function draw_horizon, with the pitch and roll used as parameters.

```
// update the aircraft display, executed from message.C
void Aircraft::updateAircraftDisplay(GUI_MSG_DATA *pdu_ptr)
{
    altitude->setDisplayValue(int(pdu_ptr->altitude));
    pitch->setDisplayValue(int(pdu_ptr->pitch));
    roll->setDisplayValue(int(pdu_ptr->roll));

    // update the attitude indicator within the GL window
    // Set the graphic window into which the gl calls will go
    GLXwinset(XtDisplay(theAttitude->baseWidget()),
        XtWindow(theAttitude->baseWidget()));

    // execute the gl code
    attitude->draw_horizon(pdu_ptr->pitch, pdu_ptr->roll);
}
```

Figure 24: Aircraft Class Function updateAircraftDisplay

I. STEALTH PANEL

The stealth panel is another of the components displayed within the instrument component. It has no user input functions that are sent to NPSNET, but instead displays information on the entity the user has selected on the radar screen. It is derived from the VkComponent class, and is instantiated from within the instrument class definition.

The stealth panel consists of a form widget as the base widget, set to the dimensions 450 by 240 pixels, to fit within the instrument panel. To this form are attached a label widget with the title "Selected Entity Information" and a scrolled window widget. The scrolled window widget is set to automatically display a scroll bar when the size of the display area exceeds the size of the scrolled window area itself.

Within the scrolled window, a row column widget organizes a series of stealth read out class instantiations (Figure 25). This class, named StealthRO, is a simple variant of the readout class, with the label to the side, a different font, and a different dimension for the

textfield widget. Due to the nature of the information displayed within the stealth window, the function setDisplayValue, found in the readout class, has been replaced with the functions setDisplayInt and setDisplayString. These are designed to display an integer or a string as required.

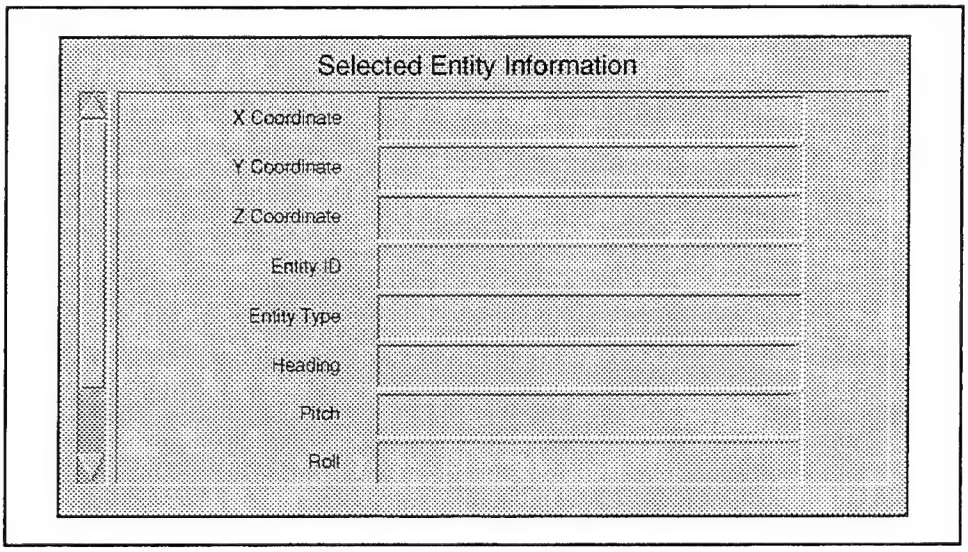

Figure 25: The Stealth Panel

The stealth panel is the only one currently within the instrument component which does not use any information from the incoming socket structures. Instead, the stealth window has a single user function, updateStealthDisplay, which accepts an integer as a parameter. This function updates all of the information fields within the scrolled window widget based upon a user's selection of a vehicle within the radar screen.

updateStealthDisplay(int) This function uses the integer parameter as an index to access the global vehicle array structure. The function is executed within the instrument function updateInstrumentDisplays. It display the appropriate information taken directly from the DIS pdu structure. First the input parameter is tested. If it is zero, the user has deselected a vehicle by clicking on an unoccupied area of the radar screen. In this case, the stealth display is cleared by calling the stealthRO function setDisplayString with a blank string for each instantiated readout (Figure 26).

```
else // no vehicle selected
{
    // clear display value if mouse is clicked with
    // no vehicle selected
    locX ->setDisplayString("No vehicle selected");
    locY ->setDisplayString(" ");
    locZ ->setDisplayString(" ");
    entityId -> setDisplayString(" ");
    entityType -> setDisplayString(" ");
    force -> setDisplayString(" ");
    headg -> setDisplayString(" ");
    pitch -> setDisplayString(" ");
    roll -> setDisplayString(" ");
    velocity -> setDisplayString(" ");
    markings -> setDisplayString(" ");
    type -> setDisplayString(" ");
    status -> setDisplayString(" ");
}
```

Figure 26: Clearing the Stealth Display

If the value is not zero, the vehicle list is accessed, the necessary information is parsed from the PDU, and the appropriate setDisplayString function is executed for each stealthRO instantiation. When necessary, the function sprintf is used to format the output correctly by copying the vehicle data into a temporary array of characters and executing the displayString function from the StealthRO class. Below is an example of several fields of the vehicle's entity information being formatted and displayed within the stealth window (Figure 27).

```
 if (index > 0)  // if a vehicle was chosen
 {
// get and display the vehicle's X, Y, and Z position
pfCopyVec3(pos,G_vehlist[index].vehptr->getposition());
locX -> setDisplayInt(int(pos[X]));
locY -> setDisplayInt(int(pos[Y]));
locZ -> setDisplayInt(int(pos[Z]));

// copy the entityID values into a string and display
entity_ID = G_vehlist[index].vehptr->getdisname();
sprintf (buffer, "%d.%d.%d",
     entity_ID.address.site,
     entity_ID.address.host,
     entity_ID.entity);
entityId -> setDisplayString(buffer);
```

Figure 27: Updating the Stealth Display

J. TANK PANEL

The tank component, derived from VkComponent, is instantiated within the instrument class, and provides input for the main tank gun azimuth (rotation) and elevation as well as a weapons panel for targeting and firing weapons systems. The digital display for gun azimuth and elevation are instantiations of the readout class. Analog feedback of the main gun azimuth and elevation is through rheostat widgets, a part of the Free Widget Foundation collection of public domain Motif widgets (Figure 28).

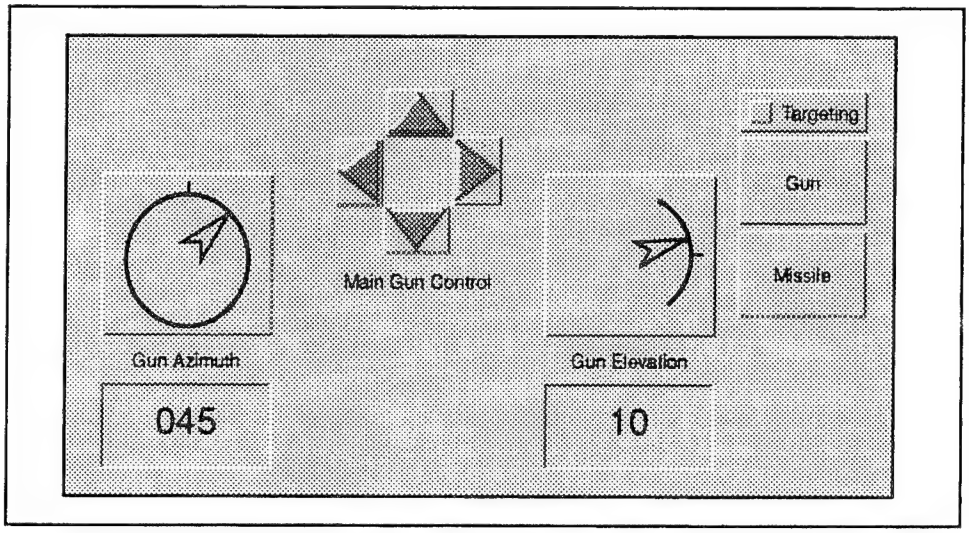

Figure 28: Tank Panel

These rheostats indicate the angle of the gun relative to the tank (Zero degrees represents the tank's heading) while elevation is relative to the tanks current pitch angle. The Rheostat widget provides, among others, parameters for setting resources for the geometry of the widget (including dial thickness and radius), color, minimum and maximum angles and input values, number of intervals, and callback procedures.

Input for the gun is a star pattern of arrow buttons created using the ViewKit class VkRepeatButton. This class modifies the behavior of any type of selected button to respond continuously to a mouse button press, rather than just on the initial mouse press event. [VIEW94]. Functions are provided to specify a callback procedure, set the initial repeat delay (in milliseconds) and the repeat rate.

updateTankDisplay(GUI_MSG_DATA) display the values of the gun azimuth and elevation rheostat widgets and their corresponding readout components. Adjustment for multiple rotations of the gun must be made, since NPSNET can generate angles greater than 360 degrees in this situation (Figure 29).

```
void Tank::updateTankDisplay(GUI_MSG_DATA *pdu_ptr)
{
    // adjust for complete and/or multiple
    // rotations of the gun
    L_gunAzimuth = int(pdu_ptr->gunAzimuth);

    while (L_gunAzimuth > 180)
    L_gunAzimuth -= 360;

    while (L_gunAzimuth < -180)
    L_gunAzimuth += 360;

    // update the azimuth display in the digital readout
    azimuth->setDisplayValue(L_gunAzimuth);

    // update the azimuth display in the rheostat gauge
    XtVaSetValues(azimuthRheostat, XtNvalue, L_gunAzimuth, NULL);

    L_gunElevation = int(pdu_ptr->gunElevation);

    // update the elevation in the digital readout
    elevation->setDisplayValue(L_gunElevation);

    // update the elevation in the rheostat gauge
    XtVaSetValues(elevationRheostat, XtNvalue, L_gunElevation, NULL);
}
```

Figure 29: Tank Class Function updateTankDisplay

K. JOYSTICK COMPONENT

The joystick component employs the only "custom written" widget used in this interface. It consists of a drawing area into which is placed a drawn button. A user places the mouse pointer over the button, pushes the left mouse button, and drags the drawn button within the area of the recessed frame (Figure 30). The movements of the drawn button represent the movements of a joystick as seen looking down from above. The top of the recessed frame represents full forward position of an actual joystick, and full motion within the X and Y axis is permitted within the frame. Upon releasing the mouse button, the joystick pops back to the center position, just as an actual spring-loaded joystick would.

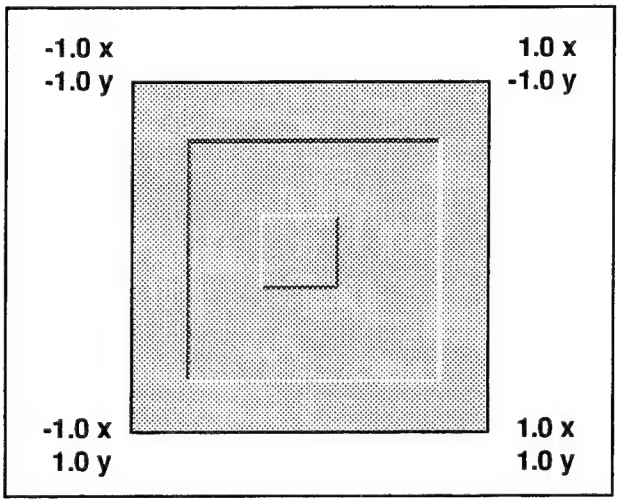

Figure 30: The Joystick Component and Coordinate System

The joystick is derived from a panner widget described in Doug Young's book *Object-Oriented Programming with C++ and OSF/Motif* [YOUN93]. The parent class is panner, with the derived class joystick adding the following user function positionToJoystick.

positionToJoystick(int positionX, int positionY) is executed on any input event to the joystick. This function converts the raw screen coordinates of the drawn button within the drawing area to values from -1.0 to 1.0 in the X and Y axis. The joystick coordinates are then adjusted to reduce the sensitivity in the center region. A message class function (joystickChanged) is then executed which formats data a packet with the new joystick data and sends it to NPSNET.

L. MAP COMPONENT

The map class provides functionality and a canvas on which to execute the external code which instantiates and draws a detailed terrain map on the screen. The code and supporting terrain database are taken from the Modular Semi-Automated Forces (ModSAF) program developed by Loral Corporation in Cambridge MA [LORA94]. The map class displays the map, and provides the user with some interaction in determining what features to display. The map is updated from within the message class timer loop.

1. ModSAF Library

The ModSAF code provides functions within the library libtactmap to initialize a terrain database, display a map of the terrain, and draw individual features on the map, including contour lines, trees, water, roads, buildings, gridlines, and more. The libraries also provide the capability to overlay dynamic vehicles and tactical features onto the map which may be access directly by user programs and functions.

The file initMap.C was adapted from the test.c program within the ModSAF library libtactmap. To avoid completely rewriting the ModSAF code, written in C, the map.C file accesses functions from initMap.c by using the command extern "C". This command allows access to standard C functions by forcing the C++ compiler to retain the original function names, rather than "mangling" them with computer-generated names during the compilation process. The following functions were defined in the file initMap.c, and used within the map class to access the ModSAF map.

void initMap(Widget, char *) initializes the graphics context of the widget passed in. The map class passes in an xmDrawingAreaWidget on which to draw the map. Since no initialize callback is provided with this widget, the expose callback was used to initialize the map. Handling events from that point on, including expose, input, and resize events, is handled by lower-level Xt functions within the ModSAF libraries. To ensure the map would be initialized only once from the interface, a flag was used within the exposeCallback function of the drawingArea widget to ignore any future expose events. Within the same initMap routine, ModSAF also sets up the overlay planes for dynamic entities, reads the internal terrain database from the terrain directory specified in the char* parameter, than creates the map and any selected features.

void updateVehicle(float posX, float posY, float hdg, float speed); This function draws a simple filled circle and a heading line onto the map, indicating the host NPSNET vehicle's current position within the virtual world. The heading line points in the direction of travel, and the length of the line indicates relative speed. It is this function which is

eventually executed from the timer loop, through function calls made from the message class to the map class and finally to the updateVehicle function.

The actual vehicle drawing, however, takes place in the function drawVehicle. This function is executed through a separate timer loop within the function initMap. In addition to drawing the vehicle's current position as a white circle overlaid onto the map, a line is drawn from the vehicle indicating the current heading, with the length of the line indicating relative speed of the vehicle. The drawVehicle function also automatically centers the vehicle when its position reaches a certain distance from the center of the map window. The decision to update the map position only periodically, rather than keeping the vehicle continuously centered in the window, was due in large part to the time required to redraw the map. Using the default terrain database for the interface panel, Ft. Benning, GA, the map redraws in less than one second. The database for Ft. Knox, KY, however, with the much larger area, greater number of features, and larger amount of data, can take from five to ten seconds to update at times.

void centerVehicle(); Centers the map display within the parent widget, using the vehicle's current coordinates as the center point of the display. The coordinates are global within the initMap.C file, so no parameters are required to be passed in.

void reduceMap(); Executes the ModSAF function tactmap_set_scale to set the scale of the map with the parameter current scale size * 2.0.

void enlargeMap(); Executes the same tactmap_set_scale with a factor of 0.5.

void addFeatures(int mapFeature); This function executes the ModSAF function tactmap_add_object with a number corresponding to the object desired. The features and their corresponding numbers are defines in the file initMap.c. This is called as a result of the user activating a toggle button down on the map display.

void removeFeatures(int mapFeature); This function executes the ModSAF function tactmap_remove_object with a number corresponding to the feature desired. This is called as a result of the user deactivating a toggle button down on the map display.

2. The Map class

The map class is primarily the interface between the ModSAF libraries and the interface panel system. The map component, derived from VkComponent, consists of a form widget, toggle buttons and pushbuttons, and a large xmDrawingAreaWidget into which the map itself is rendered (Figure 31). The only user function required updates the map given the socket_structure containing the data as a parameter.

void updateMap(GUI_MSG_DATA *) Uses the vehicle position, heading, and speed from the GUI_MSG_DATA structure to redraw the vehicle on the tactical map display. It executes the function updateVehicle from the file initMap.c to update the vehicle, passing in the parameters positionX, positionY, heading, and velocity.

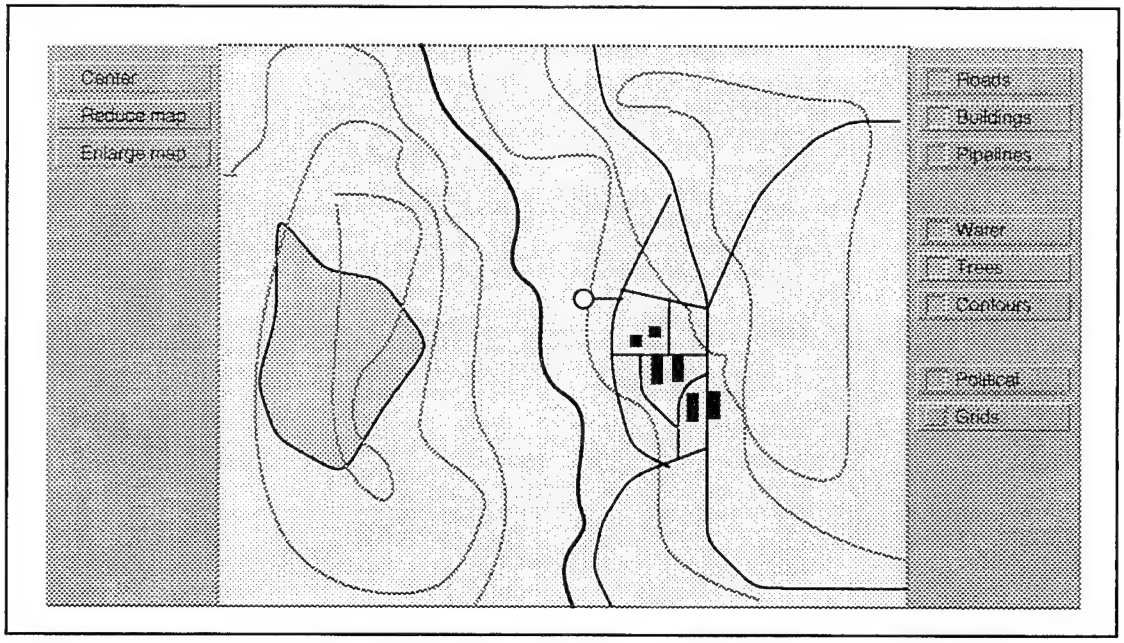

Figure 31: The Map Component

Another feature available to the user is the ability to scale and center the map by clicking the mouse over the map drawing itself. These mouse events are not intercepted at the Motif level of the interface, but rather at the lower X Windows level by the ModSAF code itself. The left button centers the map at the cursor location (note the automatic

centering of the vehicle within the map may override the user selection). Button two scales the map up, and button three scales it down.

M. THE COMPLETE INTERFACE PANEL

All of the components described thus far form a hierarchy of larger components until final placement within the interface. The top level component is the panel, which is also the main window for the interface. Placement of the main components within the panel is done in the same way that lower level components are placed within their parent components. Xt resources for the component's top level widget are set within the panel by executing each component's baseWidget function.

The panel is also responsible for setting up the pull-down menu structure and its associated callbacks. This is done through the ViewKit class VkMenu and its supporting functions (Figure 32). Callback functions for the menu items are specified in the

```
//*********************************************************
// Define menu items here.

// disable the menus being built in the work
// procedures, so the timer in the message class
// does not interfere with their creation
VkMenu::useWorkProcs(False);

// Add a menu pane "Application"
VkSubMenu *appMenuPane = addMenuPane("Application");

appMenuPane->addAction("Quit",
            &Panel::quitCallback, (XtPointer) this);

// Add a second menu pane "Vehicle"
VkSubMenu *vehicleMenuPane = addMenuPane("Panel");

VkRadioSubMenu *vehicleRadio =
            vehicleMenuPane->addRadioSubmenu("Vehicle");

vehicleRadio->addToggle("Stealth",
            &Panel::stealthCallback, (XtPointer) this);
vehicleRadio->addToggle("Aircraft",
            &Panel::aircraftCallback, (XtPointer) this);
vehicleRadio->addToggle("Tank",
            &Panel::tankCallback, (XtPointer) this);
```

Figure 32: Instantiation of the Pull-down Menus

commands addAction and addToggle. Notable here is the fact that the VkPeriodic class timer function, employed within the message class to read the sockets, was interfering with the creation of the menu system in the panel. This was due to the fact that the timer was interfering with the instantiation of the lower-level Motif work procedures, normally hidden within the ViewKit framework. A switch was used in the VkMenu class to force the use of the work procedures to instantiate the menus to be turned off.

Since the panel contains only component classes, and no pure Motif widgets, the real beauty of the object-oriented paradigm is very apparent. The panel code for each component is about as simple as it can get: instantiate the component class and set its placement resources within the panel (Figure 33). All communications and functions for each component are then handled within the respective component itself. Every interior detail of the component is hidden from the panel. It is at this highest level of the interface that the advantage and speed of using component libraries is truly realized. New designs for panels can be created just by creating the component necessary, and instantiating it within the panel. Rearrangement of components within the panel is now just a trivial matter of resetting the placement resources.

```
//*************************************************************
// Insert component classes onto the form here.
// The top level widget is "parent"

Throttle *throttle = new Throttle("throttle", parent);

// position the throttle component within the panel
XtVaSetValues(throttle->baseWidget(),
        XmNtopAttachment, XmATTACH_FORM,
        XmNtopOffset, margin,
        XmNleftAttachment, XmATTACH_FORM,
        XmNleftOffset, margin,
        NULL);

throttle -> show();
```

Figure 33: Instantiation and Placement of Components in the Panel Class

The complete panel, with all of its components, uses up an entire 1280x1024 display area of the screen. Placement of the controls and displays at the top of the screen was done to enhance its usability in the context of the Computer Graphics Laboratory at NPS, since the panel was to be used in conjuction with large projection television screens displaying the outside view of NPSNET generated from the host computer.

VI. COMMUNICATION MECHANISMS

The communications mechanisms within the interface panel can be grouped into the three basic categories of communication between classes, communication with NPSNET, and communication with the DIS network traffic (Figure 34).

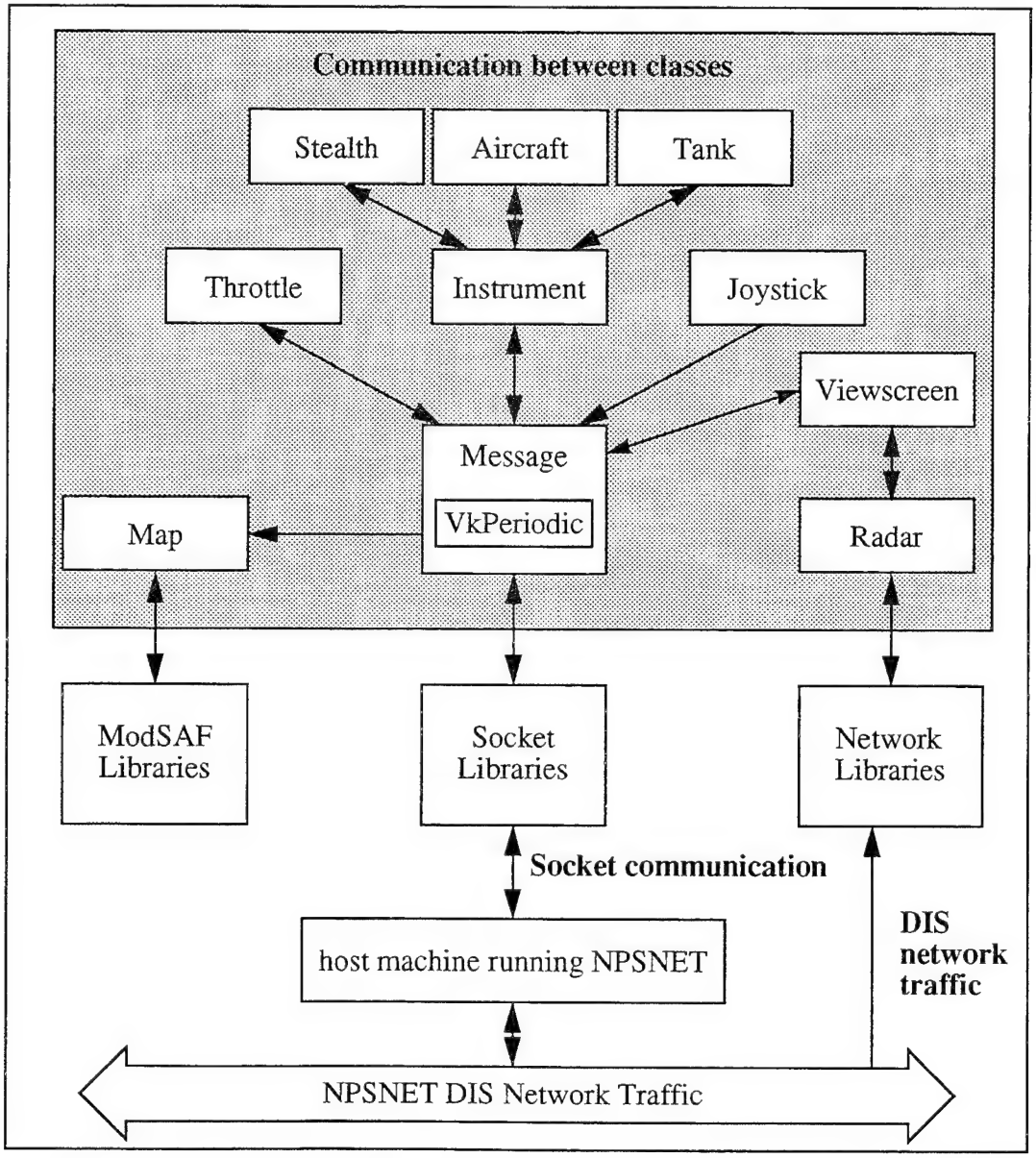

Figure 34: Overview of Message Flow within the Interface Panel

A. COMMUNICATION BETWEEN CLASSES

The emphasis in designing the communication structure between classes was to create as logical and simple a messaging system as possible. To this end, all communications from NPSNET to the interface are routed through the message class, and down a hierarchical structure to the relevant components (Figure 35). Any widget that accepts input from the user sends that information directly to the message class, where it is sent to NPSNET or distributed to other class structures.

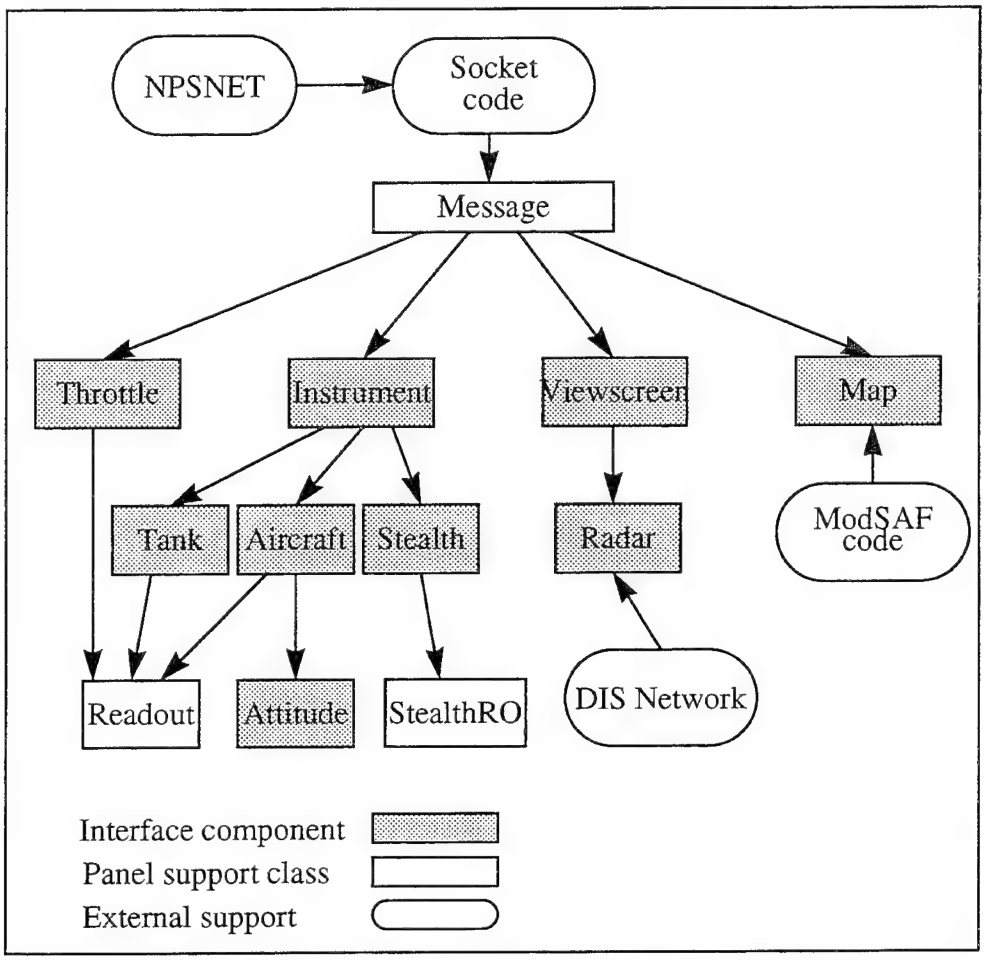

Figure 35: Data Paths from NPSNET to Components

There is one exception to this rule. This is the communication between a class structure and any outside code that supports functions within that class. The major examples of this situation within the interface panel are the connections between the map class and its

supporting code within the file initMap.c, the radar class and the DIS network support code, and the message class and the socket code in socketLib.C.

All other communication takes place to and from the class message. The primary responsibility of this class is to act as a central distribution point for information coming in from the socket connection, and a collection point for information to go to NPSNET through the socket connection.

Outgoing traffic from the interface panel to NPSNET only takes place as a result of user-initiated events on the interface itself. As each event takes place, whether it is a menu selection or the push of a button, a callback routine for that widget is executed. This callback then executes a procedure within the same class which sends the appropriate value to the message class by running a procedure within the message class itself. As an example, the flow of data is traced for an event in which the throttle scale widget is moved by the user (Figure 36).

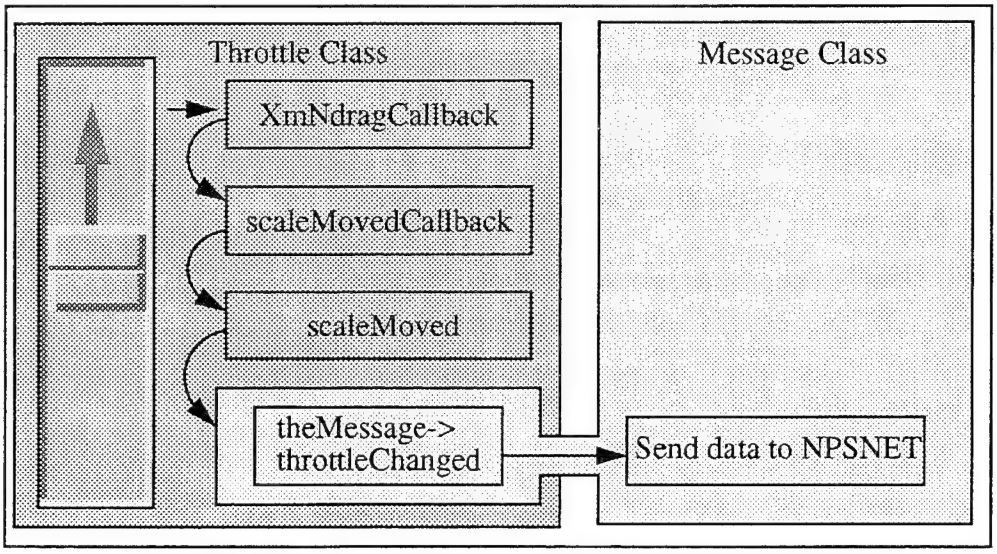

Figure 36: Example of Outgoing Communications Flow

The first function is automatically executed as a callback resource by the Motif toolkit. A callback resource for a widget is a class of resources within the Motif toolkit that allows the programmer to specify a function to execute for each type of supported interaction with

the user. The function is executed when the user performs certain actions on the widget, such as pushing a button, or moving a slider. Callbacks specify which code to execute on events that are expected to occur for that widget. So, for example, a button widget would expect to be pushed with a mouse click. When a mouse button push takes place (a click), the window manager intercepts the position of the pointer and checks to determine over which widget (if any) the event took place [HELL94].

The callback routine for this event and widget is then invoked, executing the appropriate application code to respond to that event. For our throttle scale widget, this callback function is an XmNdragCallback, which intercepts any mouse dragging events on the scale widget itself, and an XmNvalueChangedCallback, which intercepts any mouse button pushing within the widget. In the following code (Figure 37), found in the file throttle.C, any drag event or click event that takes place within the area of the throttle scale widget executes the function scaleMovedCallback.

```
XtAddCallback(scale, XmNvalueChangedCallback,
    &Throttle::scaleMovedCallback, (XtPointer) this);
XtAddCallback(scale, XmNdragCallback,
    &Throttle::scaleMovedCallback, (XtPointer) this);
```

Figure 37: Assignment of a Callback Routine

The function scaleMovedCallback then sends the scale's new value as the XtPointer callData to the functions scaleMoved (Figure 38).

```
void Throttle::scaleMovedCallback(Widget w,XtPointer,
                                  XtPointer callData)
{
    theThrottle->scaleMoved(w, callData);
}
```

Figure 38: The scaleMovedCallback Function

The function scaleMoved then passes the value of the scale position to the message class by invoking a function within the message class (Figure 39).

```
void Throttle::scaleMoved(Widget, XtPointer callData)
{
    XmScaleCallbackStruct* cbs = (XmScaleCallbackStruct*)
                                 callData;
    theMessage->throttleChanged(cbs->value);
}
```

Figure 39: The scaleMoved Function

At this point, the throttle class is done, and the responsibility for handling the event is passed on to the message class, which handles the communications between NPSNET and the NPSNET Interface Panel.

B. COMMUNICATION WITH NPSNET

As stated previously, all communications between the interface panel and the computer running NPSNET are handled by the message class using functions defined in the socket library. The message class formats and writes outgoing sockets as a result of input events from the panel. Sockets received from the host computer are read and the information is distributed to all applicable panel components.

All communications between the host computer and the panel are through a single socket structure. The organization of that structure is designed to be easily expanded if necessary. The structure is composed of many different formats, including bit masks, floating point numbers, integers, and other structures (Figure 40). It is called as the variable GUI_MSG_DATA pdu within the message.C file. The means by which the panel and NPSNET communicate is by updating and passing the structure pdu back and forth across the network to each other.

1. Communication From the Panel to NPSNET

Outgoing information is routed through the appropriate class to the message class by executing a function within the message class itself. In the previous example, the throttle

```
typedef unsigned char BYTE;

typedef struct {
    // identify the type of data structure being passed
    // To be used for future expansion, should the need
    // arise for additional data structures
    double status;
    unsigned short type;
    unsigned short length;

    // throttle data required to read the throttle position
    // (-1.0 to 1.0) and to set the throttle input with the
    // scale widget
    float throttleSetting;

    // joystick data required to read the joystick position
    // and set input as required (each is between -1.0 and 1.0)
    float joystickX;
    float joystickY;

    // vehicle settings read from NPSNET
    float positionX;
    float positionY;
    float positionZ;

    float altitude;
    float heading;
    float pitch;
    float roll;
    float velocity;

    float gunAzimuth;
    float gunElevation;

    EntityID vehicleID;

    // flag settings to execute NPSNET actions
    EntityID targetVehicleID;

    BYTE attachMode;  // including tether, attach, target,teleport

    BYTE weaponsMode; // including primary, secondary,tertiary,
                      // targetingEnable

    BYTE hudMode;     // including hudEnable

    BYTE environmentMode; // including fogEnable,wireframeEnable,
              // textureEnable, cameraEnable

    // variable settings to control NPSNET functions
    BYTE fogSetting;
    BYTE hudSetting;

} GUI_MSG_DATA;
```

Figure 40: Socket Communications Data Structure

was moved, and the chain of information flow was traced to the message class with the execution of the function

```
theMessage -> throttleChanged
```

The throttleChanged function formats a pdu structure by converting the integer parameter into a float. The output is divided by 100 because NPSNET expects a value here of between -1.0 and 1.0. The converted throttle setting is then send to the network with the following code (Figure 41).

```
// Throttle changed functions
// First set the throttle variable in GUI_MSG_DATA and then
// write the entire socket structure.

void Message::throttleChanged(int throttle)
{
    pdu.throttleSetting = float(throttle)/100.0;
    writeSocket();
}
```

Figure 41: Message Class throttleChanged Function

The function writeSocket, also within the message class, executes a network library function, socket_write, to send the newly formatted pdu to the network (Figure 41). If the socket write fails, the function flags the failure and exits the program with an error message describing the reason.

```
void Message::writeSocket()
{
    if (socket_write((char *) &pdu) != TRUE)
    {
        cerr << "socket failure, write failed" << endl;
        cerr << "Exiting program \n";
        socket_close();
        exit(0);
    }
}
```

Figure 42: Message Class writeSocket Function

2. Communications from NPSNET to the interface

Incoming sockets are handled somewhat differently than outgoing sockets. Since a socket is sent to the host computer only as a result of a recognized Motif event originating

with user interaction with the interface, the chain of events leading to the socket_write can be easily started from a Motif callback function. This is, however, not the case with an incoming socket.

No specific event flags the arrival of a socket in the same way an outgoing socket is triggered. Some mechanism was necessary to create a continuous stream of events that would trigger the distribution of all incoming sockets to the appropriate components. This stream was created using a timer function contained within the ViewKit class VkPeriodic. A timer was instantiated within the message class that generates an event every millisecond (Figure 41). This event can then be used to trigger the execution of a callback function named functionLoop in the same way any callback is invoked.

```
// set up the timer loop to read the sockets at the interval
// specified by the VkPeriodic argument ( in ms ) and
// execute the procedure functionLoop at each tick.
timer = new VkPeriodic(1);
VkAddCallbackMethod(VkPeriodic::timerCallback,
          _timer, this, &Message::functionLoop, NULL);
```

Figure 43: Instantiating the Timer Class

The callback functionLoop executes a function called readSocket on every timer event. This function readSocket checks the socket queue (within the socket library code) for any sockets. If none exist, the function returns. If the queue is not empty the information is sent to the major components by executing update functions within each component class.

For example, an incoming socket arrives and is placed in the socket queue. At the next timer event, the queue is checked. If no socket is present, the function ends. When a socket is present, the pointer to that socket, pdu_ptr, is passed to each of the major components of the interface. It is then the responsibility of each component to distribute and display the information itself (Figure 41).

Of note here is the function updateInstrumentDisplays. Unlike the other update functions, which just accept a pointer to the incoming data, the instrument display requires an additional parameter, the integer _pickVehIndex. This variable is global within the

```
void Message::readSocket()
{
     // read in any incoming sockets and distribute the
    // appropriate information throughout the panel display.

    static GUI_MSG_DATA     *pdu_ptr;
    static struct readstat  rstat;

    if ((pdu_ptr = socket_read(&rstat)) == NULL)
    {
    // no pdu's available to read
    }
    else
    {

    // update the throttle
    theThrottle->updateThrottle(pdu_ptr);

    // update the instrument component
    theInstrument->updateInstrumentDisplays (pdu_ptr,
                _pickVehIndex);

    // update the radar screen
    theViewscreen->updateScreen(pdu_ptr);

    // update the tactical map
    theMap->updateMap(pdu_ptr);
    }
}
```

Figure 44: Message Class Function readSocket

message class, and refers to the index of the vehicle within the DIS vehicle array that was selected on the radar screen. The _pickVehIndex is only used by the stealth module, and could conceivable have been updated directly by the radar class itself. In keeping with the overall design principle of using the message class as a central distribution point, the selected vehicle is updated within the message class by the function pickVehicleChanged.

By routing through the message class, the additional benefit is gained that each major module, the throttle, instrument, viewscreen, joystick, and map are responsible for their own updates, and the internal functions within each are properly hidden. This modularity is not just limited to the major components either. Within each major component, such as the instrument component, the internal components such as aircraft, stealth, and tank follow the same paradigm. The instrument component merely executes update functions within each of these, passing in the appropriate pdu_ptr or _pickVehIndex as necessary. These

"sub-components" then update themselves. The object-oriented system ensures that major and lower-level components are as independent and self-contained as possible.

C. COMMUNICATION FROM DIS NETWORK TRAFFIC

The communications between the DIS traffic and the control panel occur in only one direction--from the network to the panel. The interface in its current configuration generates no DIS traffic. This function is handled by the host computer running NPSNET.

The reception of DIS network traffic is used within the panel to update the positions of the simulation entities on the radar screen. The decision to access the net directly, rather than having the host computer send the information through the socket stream, was made to reduce the computational workload on the host computer and the required bandwidth on the network. Since the information required to update the radar screen was already available in DIS format, no advantage was gained in adding the additional burden on the host computer to translate this data into the socket format. More to the point, however, the information was already on the network. Rebroadcasting redundant information across the same network would increase the traffic substantially with no real advantage to the interface or host computer.

The network connection is opened along with the socket connection within the main.C program. This ensures that only one net and socket open command are executed within the program, and that no attempt to draw the interface occurs unless the network connections are successfully opened and initialized (Figure 41).

The DIS network library used in this module was developed by John Locke at the Naval Postgraduate School. It provides the basic commands net_open(), net_read(), net_write(), and net_close() to the user [LOCK92]. The DIS protocol was incorporated by Steve Zweswitch, making NPSNET interoperable with many other simulation systems [ZESW93]. Paul Barham rewrote the libraries in 1994 in C++ using object-oriented techniques, bringing the network code more in line with the object-oriented version of NPSNET currently employed.

```
//set up the entity arrays for radar screen
initialize_vehicle_arrays();
setup_type_table("./datafiles/dynamic.dat");
setup_entity_net_table();

// Set input buffer length (in PDUs) for sockets
if (buf_len == 0)// not set in command line
   buf_len = 32;
cerr << "Setting input buffers to " << buf_len << " PDUs\n";

// get the correct host id and network interface from the
// hosts.dat file
findsitehost(ether_interf);

// open the DIS network to receive entities
// for the Radar screen
openNetwork(ether_interf);

// open the network for socket communication with NPSNET
if (socket_open(ether_interf, buf_len) == FALSE)
{
   cerr << "main(): socket_open failed "
        <<ether_interf<<endl;
   exit(1);
}
```

Figure 45: Establishing the Network Connection

VII. RESULTS

The result of the research is a control panel interface implemented in C++ using Motif libraries. Separation of the interface component from NPSNET has been achieved through a socket-based communications link between the host computer running NPSNET and a second computer running the interface application. The interface display is housed in a simulated cockpit set up within the Computer Graphics and Video Lab at the Naval Postgraduate School, along with a throttle and joystick. This cockpit sits in front of three rear-projection television screens which display the visual output of NPSNET (Figure 46).

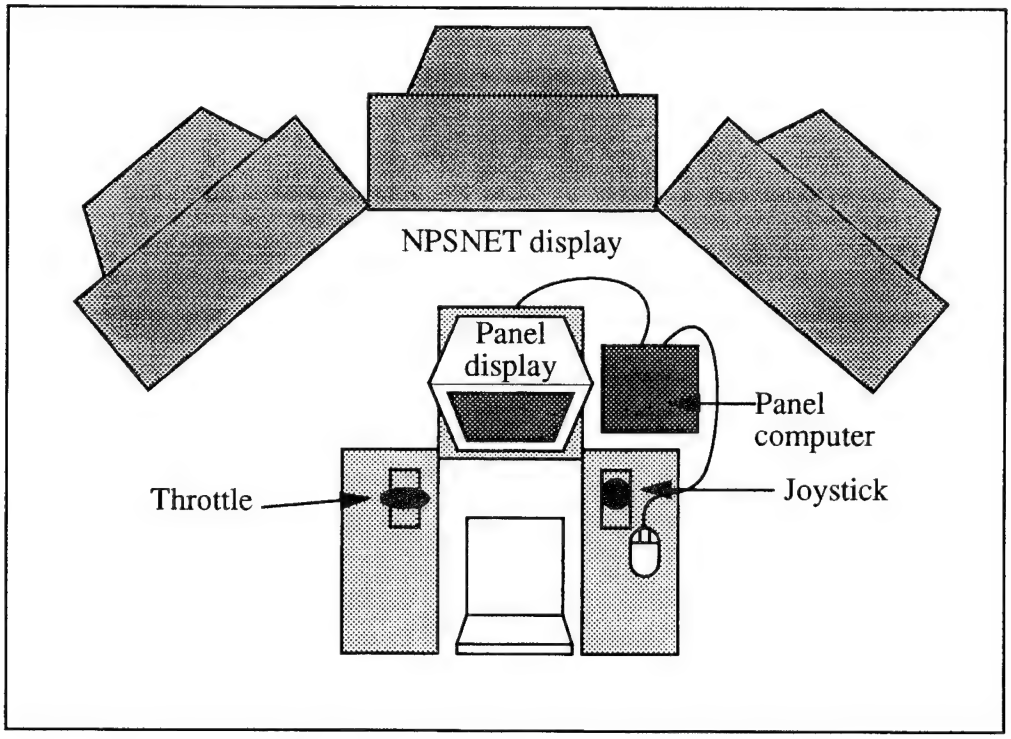

Figure 46: NPSNET Virtual Cockpit Interface Setup

Development of the control panel required minor changes to NPSNET to incorporate the socket link. Once the socket framework was in place, continued development and testing

of the interface panel required virtually no additional changes to the core NPSNET code. Thus development of additional interfaces can continue without affecting NPSNET itself.

Additionally, an applications framework has been created facilitating the rapid development and incorporation of new Motif-based interface components within the panel. This framework is based on object-oriented programming techniques which allow for the creation of independent interface modules of varying complexity that can be easily reused and refined as requirements dictate. Plans are already underway to develop new components for existing and planned vehicles and platforms within NPSNET.

VIII. CONCLUSIONS AND FUTURE AREAS OF RESEARCH

A. CONCLUSION

The objectives of this research were to develop an interface application framework for NPSNET that: (1) facilitated the rapid development and prototyping of control panel interfaces for NPSNET; and (2) separated the interface component from NPSNET. An object-oriented applications framework was developed with support for user-defined control panel components and a communications structure with NPSNET. An interface was developed and successfully employed with NPSNET using three vehicle configurations. The following conclusions have been reached:

- An effective applications framework now exists upon which new interface modules may be rapidly developed and incorporated into NPSNET.
- Separation of the interface module from the core NPSNET code has been achieved through socket-based network communications.
- The requirement for extensive use of confusing heads-up displays within NPSNET has been eliminated with use of the control panel interface.

B. FUTURE RESEARCH

This research provides a basic foundation for continued exploration of user interaction issues within NPSNET. The following is a list of topics for future research.

- Continued development and refinement of existing interface components.
- Further optimization of the communications mechanisms to enhance speed and efficiency.
- Incorporation of physically-based modelling for vehicle behavior into the interface component.
- Implementation of voice control within interface components.
- Incorporation of Performer displays within the interface for secondary viewpoints.
- Formal usability studies on interface components.
- Enhancement of component libraries.
- Separation of the simulation engine from the visualization engine in NPSNET.

LIST OF REFERENCES

[HELL94] Heller, D., Ferguson, P. M., *Motif Programming Manual,* Volume 6, O'Reilly and Associates, Sebastopol, CA 1994.

[LORA94] "ModSAF User Manual Version 1.2" ModSAF user document, Loral Advanced Distributed Simulation, June, 1994.

[NEID93] Neider, J., Davis, T., Woo, M., *OpenGL Programming Guide,* Addison-Wesley Publiching Co., Reading, MA 1993.

[VIEW94] "IRIS ViewKet Programmer's Guide" [IRIS 5.2 Electronic Reference Book]. Available throught the Silicon Graphics IRIS 5.2 Operating System online documentation.

[YOUN90] Young, D. A., *The X Window System Programming and Applications with Xt, OSF/Motif Edition,* Prentice Hall, Englewood Cliffs, NJ, 1990.

[YOUN93] Young, D. A., *Object-Oriented Programming with C++ and OSF/Motif,* Prentice Hall, Englewood Cliffs, NJ, 1992.

[ZESW93] Zeswitc, S. R., "NPSNET: Integration of Distributed Interactive Simulation (DIS) Protocol for Communication Architecture and Information Interchange", Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1993.

[ZYDA93] Zyda, M., Pratt, D., Falby, J., Barham, P., Kelleher, K., "NPSNET and the Naval Postgraduate School Graphics and Video Laboratory", *Presence* Vol. 2, No. 3, June, 1994.

INITIAL DISTRIBUTION LIST

1	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2	Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943-5101	2
3	Dr. Ted Lewis, Chairman and Professor Computer Science Department Code CS/LT Naval Postgraduate School Monterey, CA 93943-5000	1
4	Dr. David R. Pratt, Assistant Professor Computer Science Department Code CS/PR Naval Postgraduate School Monterey, CA 93943-5000	3
5	John S. Falby, Assistant Professor Computer Science Department Code CS/FJ Naval Postgraduate School Monterey, CA 93943-5000	3
6	Michael J. Zyda, Professor Computer Science Department Code CS/ZK Naval Postgraduate School Monterey, CA 93940-5000	1
7	Paul Barham, Computer Specialist Computer Science Department Code CS/Barham Naval Postgraduate School Monterey, CA 93940-5000	1
8	Christopher B. McMahan, Lieutenant 22385-B Rancho Ventura St. Cupertino, CA 95014	2